

KwikNet[®]

TCP/IP Stack

USER'S GUIDE

Version 3

First Printing: May 15, 1998

Last Printing: September 15, 2005

Manual Order Number: PN303-9

Copyright © 1997 - 2005

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5
Phone: (604) 734-2796
Fax: (604) 734-8114

TECHNICAL SUPPORT

KADAK Products Ltd. is committed to technical support for its software products. Our programs are designed to be easily incorporated in your systems and every effort has been made to eliminate errors.

Engineering Change Notices (ECNs) are provided periodically to repair faults or to improve performance. You will automatically receive these updates during the product's initial support period. For technical support beyond the initial period, you must purchase a Technical Support Subscription. Contact KADAK for details. Please keep us informed of the primary user in your company to whom update notices and other pertinent information should be directed.

Should you require direct technical assistance in your use of this KADAK software product, engineering support is available by telephone, fax or e-mail. KADAK reserves the right to charge for technical support services which it deems to be beyond the normal scope of technical support.

We would be pleased to receive your comments and suggestions concerning this product and its documentation. Your feedback helps in the continuing product evolution.

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5

Phone: (604) 734-2796
Fax: (604) 734-8114
e-mail: amxtech@kadak.com

**Copyright © 1997-2005 by KADAK Products Ltd.
All rights reserved.**

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of KADAK Products Ltd., Vancouver, BC, CANADA.

DISCLAIMER

KADAK Products Ltd. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability and fitness for any particular purpose. Further, KADAK Products Ltd. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of KADAK Products Ltd. to notify any person of such revision or changes.

TRADEMARKS

AMX in the stylized form and KwikNet are registered trademarks of KADAK Products Ltd. AMX, AMX/FS, InSight, *KwikLook* and KwikPeg are trademarks of KADAK Products Ltd. UNIX is a registered trademark of AT&T Bell Laboratories. Microsoft, MS-DOS and Windows are registered trademarks of Microsoft Corporation. All other trademarked names are the property of their respective owners.

Copyright Notice

Some components of the KwikNet TCP/IP Stack have been derived from the University of California's Berkeley Software Distribution (BSD). Some components have been adapted from software made available by the Massachusetts Institute of Technology and Carnegie Mellon University. Use of this software requires the following software copyright acknowledgements.

Copyright © 1982, 1986 Regents of the University of California All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright © 1988, 1989 Carnegie Mellon University All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of CMU not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

CMU DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL CMU BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

This page left blank intentionally.

KwikNet TCP/IP Stack User's Guide

Table of Contents

	Page
1. KwikNet Overview	1
1.1 Introduction	1
1.2 General Operation	7
KwikNet Operation	9
Multitasking Operation	10
Single Threaded Operation	11
The Single Threaded Server Queue	12
1.3 KwikNet Nomenclature.....	13
1.4 Byte Ordering and Endianness	14
1.5 Memory Allocation Requirements	16
Standard C Allocation	16
KwikNet Simple Heap	16
OS Memory Management	17
Memory Acquisition Function	17
Memory Allocation Protection.....	18
1.6 KwikNet Data Logging Service	19
Message Formatting.....	19
Message Print Attributes.....	20
KwikNet Data Log Function.....	21
1.7 KwikNet Message Recording Service.....	22
1.8 KwikNet Console Driver.....	23
Serial I/O Terminal as the Console Device	24
PC Display/Keyboard as the Console Device	24
Telnet as the Console Device.....	24
AMX Console Devices	24
1.9 Debugging Aids	25
Debug Logging	25
Programmed Halt	25
Fatal Errors	25
Breakpoint Traps.....	26
Monitoring Memory Usage.....	26
Debug Mask	27
1.10 KwikNet TCP/IP Sample Program - A Tutorial.....	28
Startup	29
Client - Server Using TCP Sockets	30
Client - Server Using UDP Sockets	32
Logging.....	33
Shutdown	33
Running the TCP/IP Sample Program	34

KwikNet TCP/IP Stack User's Guide

Table of Contents (continued)

	Page
2. KwikNet System Configuration	35
2.1 Introduction	35
KwikNet Library	35
2.2 KwikNet Configuration Builder	37
Starting the Builder	37
Screen Layout	38
Menus.....	39
Field Editing.....	40
Add, Edit and Delete KwikNet Objects	41
2.3 KwikNet Library Configuration	43
Target Parameters	44
OS Parameters.....	46
General Stack Parameters	50
IPv4 Stack Parameters	53
TCP Stack Parameters.....	56
Ethernet Parameters	59
PPP / SLIP / Modem Parameters	62
DNS Client Parameters	65
Optional Components	67
Debug and Logging Parameters	69
2.4 Adding an Ethernet Network Interface.....	73
Ethernet Device Driver Definition	75
Ethernet IP Address Definition	77
2.5 Adding a SLIP Network Interface.....	81
SLIP Serial Device Driver Definition	83
SLIP IP Address Definition	85
2.6 Adding a PPP Network Interface	87
PPP Options Dialog	90
PPP Serial Device Driver Definition.....	94
PPP IP Address Definition.....	96
2.7 Modem Options.....	99
 3. KwikNet System Construction	 103
3.1 Building an Application	103
3.2 Making the KwikNet Library	104
KwikNet Directories and Files.....	106
Getting Ready	106
KwikNet Library Make File.....	107
Gathering Files.....	107
Creating the KwikNet Library	108
3.3 KwikNet Library Compilation Output	109
3.4 Compiling Application Modules	110
3.5 Linking the Application	111
3.6 Making the TCP/IP Sample Program.....	112
TCP/IP Sample Program Directories	112
TCP/IP Sample Program Files	113
TCP/IP Sample Program Parameter File.....	114
TCP/IP Sample Program KwikNet Library	114
The TCP/IP Sample Program Make Process.....	115

KwikNet TCP/IP Stack User's Guide

Table of Contents (continued)

	Page
3. KwikNet System Construction (continued)	116
3.7 Using KwikNet with AMX	116
3.7.1 AMX System Configuration	116
KwikNet Task	116
AMX Interrupt Stack	117
KwikNet Semaphores	117
KwikNet Memory Pool.....	117
KwikNet Timer	117
KwikNet Restart and Exit Procedures.....	118
AMX 86 PC Supervisor	118
3.7.2 AMX Target Configuration.....	119
32-Bit AMX Systems.....	119
16-Bit AMX 86 Systems.....	119
3.7.3 Toolset Considerations.....	120
Tailoring Files	120
Compiler Configuration Header File.....	120
OS Interface Make File.....	120
3.7.4 AMX Application Construction Summary.....	121
 4. KwikNet Low Level Services	 123
4.1 The UDP Programming Interface.....	123
The UDP Channel	124
Receiving UDP Datagrams	125
Processing Received UDP Datagrams	126
Broadcast UDP Datagrams	126
UDP Echo Requests.....	126
4.2 DHCP, BOOTP and Auto IP.....	127
4.2.1 DHCP and BOOTP	127
DHCP and BOOTP Operation	128
DHCP Timeout	129
DHCP Leases	129
DNS Server Identification Via DHCP	129
4.2.2 Auto IP Operation	130
4.2.3 IP Address Notification.....	130
4.3 The DNS Client.....	131
DNS Servers.....	131
DNS Queries	132
Get Host By Name	132
4.4 ICMP Protocol	133
ICMP and Raw Sockets	133
Using PING.....	133

KwikNet TCP/IP Stack User's Guide

Table of Contents (continued)

	Page
4. KwikNet Low Level Services (continued)	135
4.5 KwikNet Network Interface Services.....	135
Introduction to Network Interfaces	135
Network Descriptor.....	136
Network Parameter Structure (Ethernet).....	137
Network Parameter Structure (SLIP).....	138
Network Parameter Structure (PPP).....	139
Network Attributes.....	140
Network States	141
Monitoring Network Events.....	141
Starting and Stopping KwikNet	142
4.6 KwikNet Library Services.....	143
 5. KwikNet TCP/IP Sockets	 181
5.1 Introduction to KwikNet Sockets	181
KwikNet Procedure Descriptions.....	181
KwikNet Sockets API.....	182
Socket Addresses	182
Non-Blocking Sockets	183
KwikNet Error Codes	183
5.2 Socket Types	184
Stream Socket (for TCP).....	184
Datagram Socket (for UDP).....	184
Using UDP Sockets.....	185
UDP Sockets Examples	186
5.3 Socket Options	187
Non-Standard Socket Options.....	189
TCP Protocol Options	189
5.4 KwikNet Socket Services.....	191
 6. KwikNet PPP Option	 225
6.1 Introduction to PPP	225
6.2 KwikNet PPP Configuration	227
PPP Network Definition.....	228
PPP Options	229
6.3 Using a PPP Network Interface.....	230
The PPP Client and Server.....	230
Opening a PPP Network Interface	231
6.4 PPP Authentication Parameters.....	232
Security Issues	233
6.5 Adding PPP to Your Application	237
KwikNet Library.....	237
Reconstructing Your KwikNet Application.....	238
AMX Considerations	238

KwikNet TCP/IP Stack User's Guide

Table of Contents (continued)

	Page
7. KwikNet Virtual File System	239
7.1 Introduction	239
VFS File System Structure.....	239
VFS File Names	240
VFS File Operations.....	241
VFS File Access Rights	242
7.2 Virtual File System Definition	243
File Compression	245
VFS Definition File Directives	246
Volume Base	246
Source Path	246
Source Files.....	247
Output File(s)	247
Compression Mode	248
Compression Strings	248
Tag String Case Adjustment	249
VFS String File	249
VFS Template File	249
Sector Size	250
Multiple Volumes	250
7.3 Using the VFS Generator	251
Examples.....	251
Running the VFS Generator.....	253
Compiling the VFS Data Files	256
Linking with the Virtual File System.....	256
7.4 Multiple VFS Volumes	257
7.5 VFS Service Procedures	258

KwikNet TCP/IP Stack User's Guide

Appendices

	Page
Appendix A. Reference Materials and Glossary	A-1
A.1 Reference Materials.....	A-1
Books	A-1
Internet Sources	A-1
A.2 KwikNet Glossary.....	A-3
 Appendix B. KwikNet Error Codes	 B-1
 Appendix C. KwikNet File System Interface	 C-1
C.1 Introduction	C-1
C.1.1 Treck File Systems	C-1
C.1.2 KwikNet Universal File System	C-2
C.2 KwikNet File System Parameters	C-3
C.3 Using the AMX/FS File System	C-6
C.4 Using the MS-DOS File System	C-8
C.5 Using a Custom File System	C-9
 Appendix D. KwikNet Administration Interface	 D-1
D.1 Introduction	D-1
User Definitions	D-1
User Access Rights	D-2
Customizing Administration Services.....	D-2
D.2 KwikNet Administration Parameters	D-3
 Appendix E. KwikNet Sample Program Architecture	 E-1
Console Interface	E-1
KwikNet Sample Program Operation with AMX	E-3
KwikNet Porting Kit Sample Program - Multitasking Operation	E-5
KwikNet Porting Kit Sample Program - Single Threaded Operation	E-7

KwikNet TCP/IP Stack User's Guide

Table of Figures

	Page
Figure 1.2-1 KwikNet Application Block Diagram	8
Figure 1.2-2 KwikNet Operation	9
Figure 2.1-1 Configuring KwikNet.....	36
Figure 2.2-1 Configuration Manager Screen Layout.....	38
Figure 3.2-1 KwikNet Library Construction.....	105
Figure 7.1-1 KwikNet Virtual File System Functions	241
Figure 7.2-1 VFS Definition File Sample 1	243
Figure 7.2-2 VFS Definition File Sample 2	244
Figure 7.2-3 VFS Compression String Definition	245
Figure 7.3-1 Using the KwikNet VFS Generator	252
Figure E-1 KwikNet Sample Program Procedures.....	E-2

This page left blank intentionally.

1. KwikNet Overview

1.1 Introduction

The KwikNet[®] TCP/IP Stack is a compact, reliable, high performance TCP/IP stack, well suited for use in embedded networking applications. KwikNet is an enhanced version of the Turbo Treck[™] TCP/IP Stack, a professional, high-quality networking product created by Treck Inc.

The KwikNet TCP/IP Stack includes a complete complement of protocols, some of which are optional. You can readily tailor the KwikNet stack to accommodate your needs by using the KwikNet Configuration Builder, a Windows[®] utility which makes configuring KwikNet a snap. Your KwikNet stack will only include the features required by your application.

KwikNet is best used with a real-time operating system (RTOS) such as KADAK's AMX[™] Real-Time Multitasking Kernel. However, KwikNet can also be used in a single threaded environment without an RTOS.

When used with the AMX multitasking kernel, KwikNet is delivered to you ready for use on a particular target processor with any of the software development tools which KADAK supports for that target. You can concentrate on your application, knowing that the integration of KwikNet with AMX is fully functional. **No porting** is required.

KwikNet can also be provided in a form most suitable for porting to your own operating system, target hardware and software development tools. The KwikNet Porting Kit permits KwikNet to be used with your own in-house RTOS or with the commercial RTOS of your choice. The kit includes an RTOS example illustrating the use of KwikNet with a custom RTOS and three examples of single threaded use: one for MS-DOS, one for the Tenberry DOS/4GW DOS Extender and one for a custom operating system. Detailed porting instructions are provided in the KwikNet Porting Kit User's Guide.

This manual makes no attempt to describe TCP/IP, what it is or how it operates. It is assumed that you have a working knowledge of the TCP/IP protocol suite as it applies to your needs. Reference materials are provided in Appendix A.

The purpose of this manual is to provide the system designer and applications programmer with the information required to properly configure and implement a networking system using the KwikNet TCP/IP Stack. It is assumed that you are familiar with the architecture of the target processor. It is further assumed that you are familiar with the rudiments of microprocessor programming including the concepts of code, data and stack separation.

KwikNet is available in C source format to ensure that regardless of your development environment, your ability to use and support KwikNet is uninhibited. The source program may also include code fragments programmed in the assembly language of the target processor to improve execution speed.

The C programming language, commonly used in real-time systems, is used throughout this manual to illustrate the features of KwikNet.

Installation

KwikNet is delivered to you on a CD-ROM. During installation you will need your KwikNet CD serial number and product installation key, both of which are recorded on a label attached to the CD-ROM case. The CD serial number is also printed on the face of the CD-ROM.

The CD serial number identifies your KwikNet CD-ROM. The product installation key identifies the specific parts which you are entitled to install from the CD-ROM.

KwikNet is installed by running the InstallShield® *SETUP.EXE* program located in the root directory of the CD-ROM. From the Windows Start, Run... menu, type *D:\SETUP.EXE* (where *D:* is your CD-ROM drive letter) and press Enter. Alternatively, browse the root directory of the CD-ROM and double click on the *SETUP.EXE* filename or icon.

The setup utility will lead you through the installation process. You may be requested to identify the software development tools which you plan to use for development. If so, only the tools specifically supported by KADAK will be listed.

The installation process will copy the product files into a directory of your choice on the disk drive of your choice. KwikNet files will be installed in subdirectory *KNTnnn* where the number *nnn* comes from the KADAK part number *PNnnn-2* used to identify KwikNet. Throughout all KwikNet manuals, the installation directory will be referred to as *KNTnnn*.

Note

The standard distribution of the Turbo Treck TCP/IP Stack from Treck Inc., as delivered to you with KwikNet, must be installed in a directory named *TRECK*.

Manuals

This **KwikNet User's Guide** describes how KwikNet is used with the standard distribution of Turbo Treck from Treck Inc. It also provides guidance for the correct use of KwikNet with the AMX RTOS.

The **KwikNet Porting Kit User's Guide** describes how KwikNet can be ported for use with any RTOS or for single threaded use. The guide describes how to adapt KwikNet for your particular choice of target processor and how to use your software development tools to construct your KwikNet application.

The **KwikNet Device Driver Technical Reference Manual** describes the KwikNet device driver interface for Ethernet and serial networks. The use of the KwikNet Modem Driver and serial device drivers with SLIP or PPP networks is also described in that manual.

The standard **Treck TCP/IP User Manual** from Treck Inc. is provided in Adobe® Acrobat® PDF file format on the CD-ROM. The manual describes the Turbo Treck TCP/IP Stack, its methods of use, its application programming interface (API) and many of the optional protocols which it supports.

Note

Throughout this manual the term RT/OS is used to refer to any operating system (OS), be it a multitasking RTOS or a single threaded OS.

The KwikNet User's Guide

Chapter 1 provides an overview of the KwikNet TCP/IP Stack. The general operation of KwikNet is described and the nomenclature used by KADAK is introduced. Appendix A includes a glossary which will help when you are stuck trying to remember what one of the many protocol mnemonics means. A number of topics unrelated to network issues are covered in this chapter. KwikNet memory allocation requirements are examined. KwikNet data logging, message recording, console support and debugging features are also presented. Finally, the KwikNet TCP/IP Sample Program used to exercise KwikNet and illustrate its proper use is described in a tutorial fashion.

Chapter 2 is your system configuration guide. You may wish to read this chapter to learn how easy it is to use the KwikNet Configuration Builder to customize KwikNet for your use.

Chapter 3 describes how KwikNet is configured for use and combined with your application to form an executable load module. It also describes how users of KADAK's AMX Real-Time Multitasking Kernel must adapt their AMX configuration for use with KwikNet. If you are porting KwikNet to your own operating environment, the material provided in this chapter will simply augment the more detailed description presented in the KwikNet Porting Kit User' Guide.

Chapter 4 presents the KwikNet network and IP/UDP application programming interface (API). This chapter includes descriptions of the lower level (IP and UDP) services which are available for applications which choose not to use the TCP protocol and its socket interface. Topics such as the UDP programming interface, the DHCP and DNS clients and PING are also covered. Network management services for adding new networks and opening and closing existing networks are described in this chapter. It also describes common utility procedures which are available for use by applications and device drivers.

Chapter 5 presents the KwikNet socket application programming interface (socket API). It includes an alphabetic summary of all the KwikNet socket procedures at your disposal.

Chapter 6 presents the KwikNet PPP Option. This optional KwikNet component is an implementation of the Point-to-Point Protocol (PPP) commonly used with serial network interfaces. This chapter describes how to incorporate and use PPP in your application.

Chapter 7 describes the KwikNet Virtual File System (VFS), its API and the VFS Generator which creates the VFS image for incorporation into your application. The VFS is a read-only file system which is well suited for use with protocols like HTTP as implemented by the KwikNet Web Server.

Other optional KwikNet components (FTP, SMTP, SNMP, TFTP, Telnet, Web Server etc.) are described in separate manuals. For some of these protocols, the programming interface is described in the Treck TCP/IP User Manual or its companion documents. These optional components will be of interest only if you have purchased and are using the relevant KwikNet options.

The Treck TCP/IP User Manual

The Treck TCP/IP User Manual provides extensive documentation of the Turbo Treck TCP/IP Stack and the many optional protocol components which it supports. Much of the documentation is tutorial in nature and for this reason deserves careful reading.

Chapter 1 provides an excellent introduction to network protocols, what they are and how they are used. Chapter 2 introduces the sockets interface commonly used for programming using the UDP and TCP protocols.

Chapters 3 and 4 of the Treck TCP/IP User Manual are devoted to the porting and configuration process. With KwikNet, the porting has been done and the components which you wish to use are ready for use. All of the application interfaces required by the Turbo Treck TCP/IP Stack are provided by KwikNet. The choices of components and features to be used in your application are easily made using the KwikNet Configuration Builder with its built-in help service to guide you to the proper selection.

If you do plan to use KwikNet with your own RTOS or single threaded system on a particular target processor with your own software development tools, the KwikNet Porting Kit makes it easy. You can simply adapt the sample KwikNet RT/OS interface for your use. And chances are that KwikNet already includes board support and compiler support for your target and tools. In any case, you do not need to know how KwikNet and the Treck TCP/IP Stack actually operate in order to adapt KwikNet for your use.

Chapter 5 presents the Programmer's Reference, the application programming interface (API) provided by the Turbo Treck TCP/IP Stack. Note that the standard BSD sockets API described in this chapter is accessed from KwikNet as described in Chapter 5 of the KwikNet User's Guide. Treck offers a number of sockets extensions and callback services which are only accessible via the Treck sockets API. Treck also provides an extensive API for access to features managed by the Address Resolution Protocol (ARP).

A significant portion of Chapter 5 is devoted to Treck initialization, timer management, device driver services and Ethernet, SLIP and PPP network management services. KwikNet's network management and device driver interfaces utilize these functions on your behalf. Hence, all but the very few services of this type listed in Chapter 4.5 of the KwikNet User's Guide can be safely ignored by your application.

Treck provides a comprehensive set of replacement functions for the standard C runtime library, some of which are documented in Chapter 5. However, these functions are not used by KwikNet. The standard C runtime library is used by KwikNet to avoid duplication of code in the memory footprint.

Chapter 6 describes a number of services offered by the Turbo Treck TCP/IP Stack, many of which are provided by optional components. KwikNet includes support for PING, file services and network statistics dumps as standard features. PING services are accessed using the Treck API. The optional KwikNet DNS Client, if enabled, is automatically initialized and configured when KwikNet starts. DNS services are then available via the Treck API.

The KwikNet Universal File System (UFS) Interface gives Turbo Treck access to all of the file systems supported by the UFS. You can therefore safely ignore the Treck file system API specification in Chapter 6.

KwikNet provides access to all of the network statistics maintained by the Turbo Treck TCP/IP Stack and its optional components. You can therefore safely ignore the Treck network statistics logging API presented in Chapter 6.

The KwikNet TFTP and Telnet clients and servers are not derived from the equivalent Treck components. Hence the description of the Treck TFTP and Telnet options in Chapter 6 can be safely ignored.

The KwikNet SMTP client and server are not derived from the equivalent Treck components. Hence the Treck SMTP Client manual can be safely ignored.

Chapter 7 describes a number of protocols which the Turbo Treck TCP/IP Stack supports, many of which are provided by optional components. KwikNet includes dialer services as a standard feature. KwikNet also includes support for DHCP and BOOTP for systems that only incorporate IPv4. For such systems, DHCP and BOOTP services are automatically employed for networks configured to require their use. An optional KwikNet component is available to provide DHCP support for systems that use both IPv4 and IPv6.

KwikNet includes modem support for serial networks. For modems requiring dialing services, the KwikNet modem driver provides dialer scripting services which are compatible with the Treck dialer API. You can therefore safely ignore the Treck dialer API presented in Chapter 7.

KwikNet supports Auto IP, IGMP, NAT and PPP, all of which are optional components. Auto IP negotiation is automatically employed for Ethernet networks configured to use the feature. IGMP and NAT, if enabled, are automatically initialized and configured when KwikNet starts. IGMP and NAT services are then available via the Treck API.

KwikNet has complete support for PPP networks. The use of PPP is greatly simplified by the KwikNet Configuration Builder which lets you predefine a PPP network and specify all of the PPP options which must be negotiated when the network is opened. By so doing, you can safely ignore most of the Treck PPP services presented in Chapter 7.

Appendices A and B in the Treck TCP/IP User Manual can be safely ignored. In particular, ignore the section of Appendix B which describes a method of using KADAK's AMX 86 RTOS with the Turbo Treck TCP/IP Stack **without** KwikNet. **Appendix C** provides useful hints for debugging.

The Treck TCP/IP User Manual describes how to port the Turbo Treck TCP/IP Stack for use with any RTOS and how to configure it for your own use. The Treck stack is ready for use with KwikNet. **The Treck port has been done.** Furthermore, the KwikNet Configuration Builder can be used to configure KwikNet for your system without having to edit the Treck *TRSYSTEM.H* header file.

1.2 General Operation

The KwikNet TCP/IP Stack and your application operate as illustrated in Figure 1.2-1. If you are using KwikNet with AMX, all of the components shown in the block diagram are provided with KwikNet, ready to use with AMX. You simply provide the application.

If you are using the KwikNet Porting Kit to port KwikNet to your operating environment, then the shaded blocks indicate modules which will require modification to adapt KwikNet for use with your application. As you can see, very few modules require adaptation.

The KwikNet TCP/IP Stack sits between your application and the network. In some cases, your application may exclude the TCP stack and interface directly with the UDP and IP stack using low level KwikNet services. In most cases, your application will use the KwikNet TCP or UDP socket services. The KwikNet application interface shields you from any direct involvement with the underlying network, device drivers or operating system.

KwikNet includes an operating system interface which makes it suitable for use with or without a real-time operating system. KwikNet is connected to the RT/OS by an OS Interface Module, a C file containing procedures which provide access to the services of the RT/OS. This module is incorporated into the KwikNet Library so that it is always available for use by your application.

The KwikNet TCP/IP Stack consists of a single KwikNet Library built according to your specifications to meet your particular needs. The stack interacts directly with one or more KwikNet device drivers, each of which connects KwikNet to a particular network. Networks can be predefined and/or dynamically constructed at runtime. Each predefined network and its associated device driver is identified in the KwikNet network configuration file *KN_NCF.C* in the KwikNet Library. The KwikNet Library is built to meet your requirements as recorded in a parameter file generated by the KwikNet Configuration Builder (see Chapter 2).

KwikNet communicates with an external network through the device driver which handles the hardware device physically connected to the network. The KwikNet device driver interface is described in the KwikNet Device Driver Technical Reference Manual. In most applications, the device driver is interrupt driven. Only in the simplest of systems can the device driver afford to use a polling strategy. The device driver interface allows KwikNet to call the driver to initiate transmissions on the network. It also allows the driver to signal KwikNet upon receipt of a packet from the network. A separate board driver connects KwikNet, its device drivers and the OS Interface Module to your target hardware in an RT/OS independent manner.

Figure 1.2-1 also shows an application OS interface, a C module used by KADAK to provide a standard interface between the RT/OS and the sample programs (applications) provided with KwikNet and its options. If you port the KwikNet sample programs (and it is recommended that you do so), you will have to use this module. You may also find that portions of this module can, with very little adaptation, be used by your own application.

Finally, the RT/OS must provide a timing source. Although the RT/OS clock driver is shown as a separate component, it may be implemented as an interrupt service routine which resides in the OS Interface Module or in the application OS interface. When AMX is used, your AMX clock driver will generate the fundamental timing needed by KwikNet.

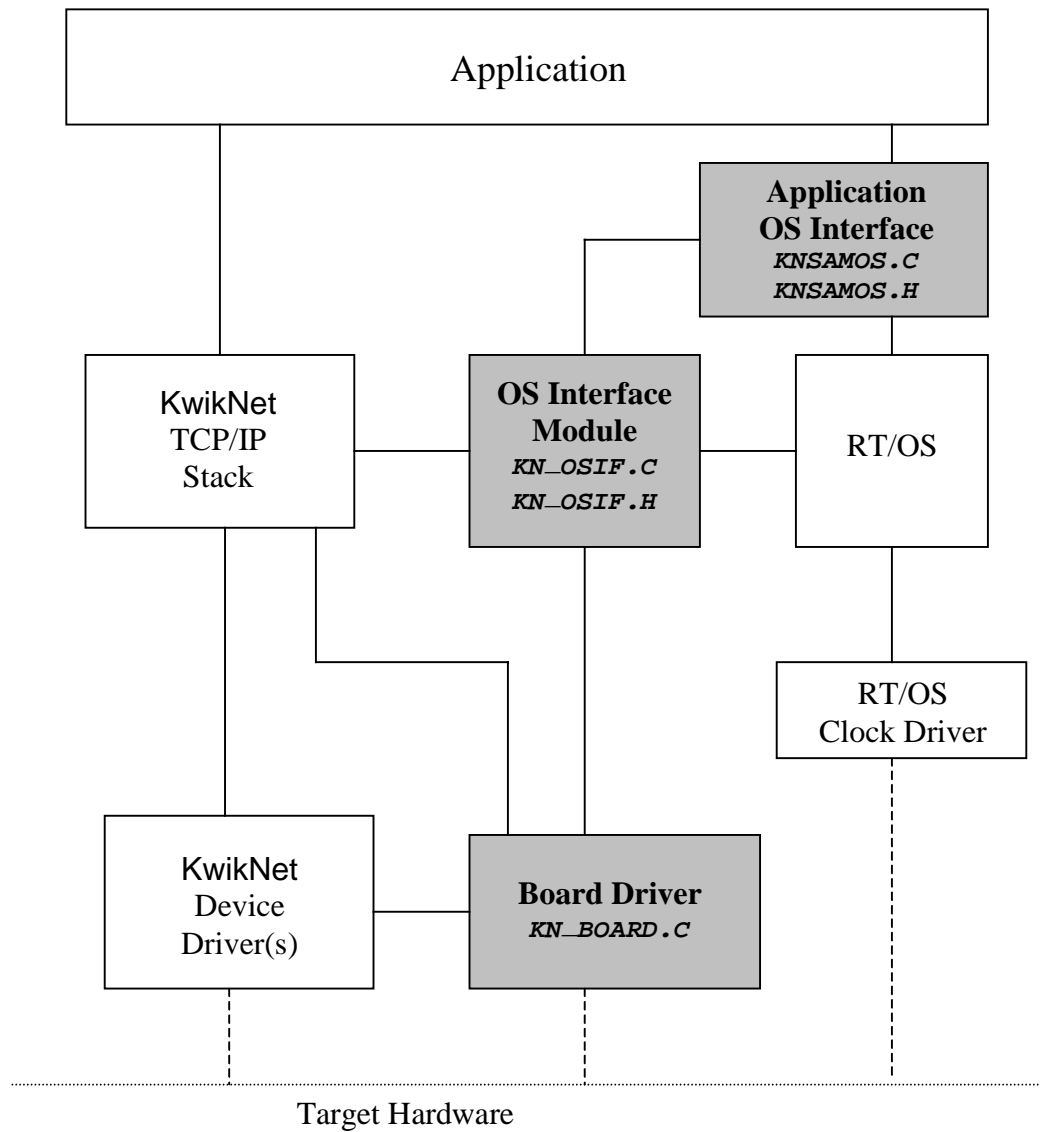


Figure 1.2-1 KwikNet Application Block Diagram

KwikNet Operation

The KwikNet TCP/IP Stack can be used with either a multitasking RTOS or a single threaded operating system. KwikNet and your application operate together as illustrated in Figure 1.2-2. Although the application interface with KwikNet is the same in both cases, the way it executes is quite different.

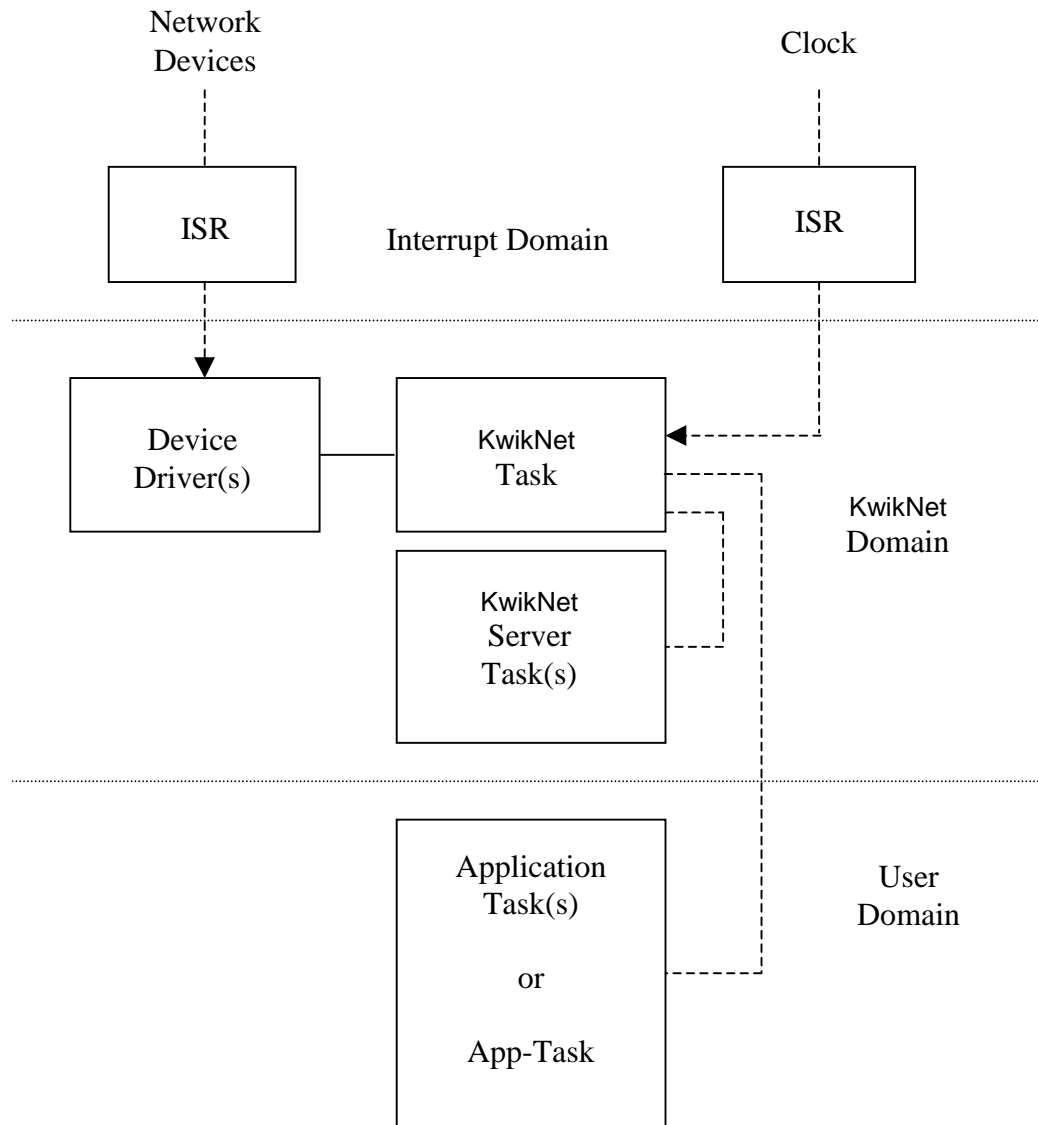


Figure 1.2-2 KwikNet Operation

Multitasking Operation

In a multitasking system which uses an RTOS, operation of the TCP/IP stack is controlled by a single task called the KwikNet Task. This task begins execution after your application calls procedure `kn_enter()` to start KwikNet. The KwikNet Task executes at a priority above that of all other tasks which use KwikNet services.

The KwikNet Task receives timer ticks from the RTOS through the KwikNet OS interface. These ticks, KwikNet's fundamental timing source, occur at the frequency which you specify when you configure your KwikNet Library.

KwikNet uses its Ethernet, SLIP or PPP network driver to interact with a particular network's device driver. The KwikNet Task and the device driver cooperate to ensure that network packet transmission and reception occur in a timely fashion. Interrupts generated by the device's hardware interface are serviced by an RTOS compatible interrupt service routine (ISR) which calls the device driver's interrupt handler.

Your application tasks interact with KwikNet using the UDP or IP programming interface or the TCP or UDP socket services.

At times, KwikNet may be forced to suspend your application task pending completion of a requested service. How this is done depends on the *magic* of the KwikNet OS interface. In a multitasking environment, only the task requesting service is suspended. Other tasks are free to execute and use KwikNet services. KwikNet and its OS interface resolve the problems, if any, which may occur when multiple tasks make conflicting demands on the use of the stack.

If you use any of the optional KwikNet components such as the FTP Server or Web Server, you will observe that these servers are also implemented as tasks running under the auspices of the RTOS. These tasks will be of lower priority than the KwikNet Task but will usually be of higher priority than your application tasks which use KwikNet services.

Finally, note that most applications will probably include one or more tasks of higher priority than the KwikNet Task. These tasks, although critical for the success of your application, must not starve the KwikNet Task's demands for execution time.

Note

All application tasks which use KwikNet services **MUST** execute at a priority below that of the KwikNet Task.

Single Threaded Operation

Single threaded operation is only supported by KwikNet if you are using the KwikNet Porting Kit.

In a single threaded system, there is a single application task which, for reference purposes, is called the **App-Task**.

The App-Task starts in your *main()* function and executes in what will be referred to as the **user domain**. Once the App-Task starts, the thread of execution is sequential, flowing back and forth between your application and KwikNet. When KwikNet (or Treck) code is being executed, your application is said to be in the **KwikNet domain**.

An interrupt can occur while executing in either the user domain or the KwikNet domain. When an interrupt occurs, an interrupt service routine (ISR) begins execution in what is called the **interrupt domain**. All interrupts, even if nested, are serviced in the interrupt domain. When service of an interrupt is finished, execution resumes in the domain which was in effect when the interrupt occurred.

Operation of the TCP/IP stack is controlled by a single body of KwikNet code which, by definition, executes in the KwikNet domain. This body of code is called the KwikNet Task, to distinguish it from your App-Task.

Your App-Task controls the flow of execution within your application. The KwikNet Task can only execute when your App-Task permits. The KwikNet Task does not begin until your application calls procedure *kn_enter()* to start KwikNet.

Once KwikNet has been started, your App-Task must regularly call KwikNet procedure *kn_yield()* to let the KwikNet Task service the TCP/IP stack. Procedure *kn_yield()* is included in the KwikNet Library and is described in Chapter 4.6.

The KwikNet Task receives timer ticks from the clock ISR through the KwikNet OS interface. These ticks, KwikNet's fundamental timing source, occur at the frequency which you specify when you configure your KwikNet Library. For best performance, your App-Task should yield to the KwikNet Task at this frequency or higher.

KwikNet uses its Ethernet, SLIP or PPP network driver to interact with a particular network's device driver. The KwikNet Task and the device driver cooperate to ensure that network packet transmission and reception occur in a timely fashion. Interrupts generated by the device's hardware interface are serviced by an interrupt service routine (ISR) which calls the device driver's interrupt handler.

Warning

Your application **MUST** regularly yield to the KwikNet Task by calling procedure *kn_yield()*. Failure to yield at least at the defined KwikNet clock frequency may result in poor performance of the TCP/IP stack.

Your App-Task can interact with KwikNet using the UDP or IP programming interface or the TCP or UDP socket services.

At times, KwikNet will be forced to suspend your App-Task pending completion of a requested service. How this is done depends on the *magic* of the KwikNet OS interface which ensures that the KwikNet Task continues to service the TCP/IP stack while waiting for the event of interest. Execution continues in the KwikNet domain until the service completes and KwikNet returns to the App-Task in the user domain.

If your App-Task makes requests for non-blocking data transfers across high speed Ethernet networks, it must increase the frequency at which it yields to the KwikNet Task in order to achieve optimum network performance.

The Single Threaded Server Queue

In a single threaded system, the KwikNet Task maintains a server queue to support the optional KwikNet components such as the FTP Server or Web Server. The KwikNet Task periodically executes each of the servers on its server queue, thereby letting these servers operate much as they would in a multitasking system.

Your application can append its own servers to the KwikNet server queue, adding a very primitive form of non-preemptive multitasking to an otherwise single threaded system.

The KwikNet TCP/IP Sample Program illustrates the process. The App-Task calls procedure `kn_addserver()` to add a TCP server to the KwikNet Task server queue. The App-Task then acts as a client using TCP to communicate with the server. When the sample is finished, the TCP server removes itself from the server queue.

Procedure `kn_addserver()` is included in the KwikNet Library and is described in Chapter 4.6.

1.3 KwikNet Nomenclature

The following nomenclature standards have been adopted throughout this manual.

Numbers used in this manual are decimal unless otherwise indicated. Hexadecimal numbers are indicated in the format *0xABCD*.

Read/write memory is referred to as RAM. Read only memory (non-volatile storage) is referred to as ROM.

KwikNet symbol names and reserved words are identified as follows:

<i>kn_pppp</i>	KwikNet C procedure name <i>pppp</i>
<i>knxtttt</i>	KwikNet structure name of type <i>tttt</i>
<i>xttttyyy</i>	Member <i>yyy</i> of a KwikNet structure of type <i>tttt</i>
<i>KN_ssssss</i>	Reserved symbols defined in KwikNet header files
<i>KN_EReeee</i>	KwikNet Error Code <i>eeee</i>
<i>KN_WReeee</i>	KwikNet Warning Code <i>eeee</i>
<i>KN_FEeeee</i>	KwikNet Fatal Error Code <i>eeee</i>
<i>KN_FFFFF.xxx</i>	KwikNet reserved filenames
<i>KNnnnnFFF.xxx</i>	KwikNet target and toolset specific filenames
<i>KNZZZFFF.xxx</i>	KwikNet filenames for application portability
<i>KN_LIB.H</i>	KwikNet Library Header File
<i>TM_FFFFF</i>	Treck macro names
<i>TM_EXXXX</i>	Treck error codes
<i>tfxxxxxxx</i>	Treck functions names
<i>ttxxxxxxx</i>	Treck type names
<i>tsxxxxxxx</i>	Treck structure names

The *nnn* in a KwikNet filename is the 3-digit KwikNet part number used by KADAK to identify a particular version of KwikNet. For example, file *KN713CM.EXE* is the KwikNet Configuration Manager provided with the KwikNet Porting Kit which is identified by KADAK part number 713.

Files with names of the form *KNZZZFFF.xxx* are intended to make KwikNet less sensitive to the environment in which it is used. For example, the KwikNet compiler configuration header file *KNZZZCC.H* is used to identify the particular characteristics of the compiler being used to construct your KwikNet application.

File *KN_LIB.H* is the KwikNet Library Header File, an include file which includes the subset of KwikNet header files needed for compilation of your application C code. By including file *KN_LIB.H* in your source modules, your KwikNet application becomes readily portable to other target processors.

Throughout this manual examples are provided in C. Code examples are presented in lower case. File names are shown in upper case. C code assumes that an *int* is 32 bits on 32-bit processors or 16 bits on 16-bit processors as is common for most C compilers.

1.4 Byte Ordering and Endianness

To use a TCP/IP stack, you must adhere to the byte ordering rules defined by the TCP/IP protocol suite. Doing so is complicated by the fact that not all target processors follow these rules.

The TCP/IP network uses the big endian model for byte ordering. Sequential bytes in the data stream always appear in sequential bytes in memory. The initial byte of the data stream is at the lowest memory address and the final byte is at the highest memory address. The network over which the transfer takes place is said to be big endian.

If a multi-byte value appears within the TCP/IP data stream, the most significant byte of that data value will always appear first in the stream. When stored in memory, the most significant byte of a multi-byte value will always appear at the lower memory address with successive bytes of the value stored at sequential, higher memory addresses. Data which resides in memory in this fashion is said to be in net endian form.

When TCP/IP is used on a processor such as the Motorola 68xxx, the network endianness matches the processor endianness. Both are big endian. Consequently, the natural storage of multi-byte values in memory matches that required by the TCP/IP data stream.

However, when TCP/IP is used on a processor such as the Intel 80x86, the network endianness conflicts with the processor endianness. The network requires big endian values in the data stream but the processor's natural storage of multi-byte values in memory is little endian.

Fortunately the KwikNet TCP/IP Stack can be used on target processors which are either big or little endian. The endianness of the target processor is a configuration parameter in the KwikNet Network Parameter File used in the construction of the KwikNet Library as described in Chapter 2.3.

Although KwikNet may be used with either big or little endian processors, it does not alleviate your application from the responsibility for correct presentation or interpretation of data delivered over the TCP/IP network. KwikNet treats your application data as a byte stream with no particular endian characteristics. It is up to your application to present the data for delivery to a remote destination in a byte ordered format that the remote end can understand.

For example, two little endian machines can send and receive data streams containing multi-byte values ordered in little endian form. The multi-byte values can be directly fetched from or stored into memory. However, if one of these machines is replaced by a big endian machine, suddenly the application will fail even though KwikNet will continue to successfully deliver the data streams between the two machines. Of course, the newer big endian machine could be reprogrammed to properly store and retrieve the little endian values expected by the other machine.

This example illustrates the absolute need for applications to agree upon the manner in which data values will be delivered to each other. Conventional wisdom suggests that if multi-byte values are always stored in net endian form, then any machine can participate in the conversation, regardless of the machine's endianness.

Net Endian Data

KwikNet provides a set of macros (or functions) which can be used by applications to convert 2-byte (short) values and 4-byte (long) values between net endian form and host endian form. These macros assume the host endianness defined in the KwikNet Network Parameter File.

The following macros are available in the KwikNet Library.

```
nlv = htonl(hlv)   Convert long from host to network endian form
nsv = htons(hsv)   Convert short from host to network endian form
hlv = ntohl(nlv)   Convert long from network to host endian form
hsv = ntohs(nsv)   Convert short from network to host endian form
```

On big endian machines, these macros leave the data parameter unaltered since the network is also big endian. On little endian machines, these macros reverse the order of the bytes in the macro parameter.

It should be obvious that *htonl* and *ntohl* are equivalent as are *htons* and *ntohs*. So which macro should be used if two are identical? Although it does not matter, it is recommended that the macro be chosen for best meaning in the context of its use. For example, when storing a *long* value *hlv* into memory for delivery in the data stream, use *htonl*(*hlv*) to indicate the conversion of the data from host to net endian form. Similarly, when fetching a *long* value *hlv* from a received data packet, use *ntohl*(*nlv*) to indicate the conversion of the data from net to host endian form.

So how does your application know which data values require conversion? There is no simple answer. The content of the data portion of any packet delivered on the network is known only to the sender and receiver. Both must agree to the method of interpretation.

Of greater concern is the management of the data while it is under the control of your application. Most hosts prefer to operate with data in the natural form dictated by the target processor. For this reason, data is usually converted to net endian form prior to sending and from net endian form upon receipt.

There are several data values which applications frequently use which, by convention, are always maintained in net endian form. These are network parameters such as IP addresses, subnet masks and default gateway addresses. It is good programming practice to always use comments to identify variables which are assumed to be kept in net endian form. All other variables can then safely be assumed to be in host endian form.

1.5 Memory Allocation Requirements

KwikNet must be able to dynamically allocate and free blocks of memory of varying sizes. KwikNet supports three memory allocation strategies using standard C, a simple heap or OS memory management services.

The KwikNet Library must be configured to select one of the three memory allocation strategies. The strategy is defined by the parameters in your Network Parameter File. The choices are made on the OS property page using the KwikNet Configuration Builder (see Chapter 2.3).

When KwikNet is used with the AMX Real-Time Multitasking Kernel, any of the strategies can be used. The examples provided with the KwikNet Porting Kit support standard C, but can be easily modified to use the small heap or the memory allocation services in your RT/OS.

Standard C Allocation

The first memory allocation strategy uses standard C library functions *malloc()* and *free()* to allocate and free memory. This strategy is best used in single threaded systems which are inherently thread-safe. If you use this strategy within a multitasking system, you will have to use KwikNet's locking service to protect access to the library functions unless your C runtime library provides thread-safe implementations of its memory management functions.

KwikNet Simple Heap

The second memory allocation strategy offered by KwikNet is its simple heap, a region of memory from which KwikNet carves memory as required. The memory region can be assigned in three ways: 1) a single static array in memory; 2) a collection of one or more blocks of memory acquired by KwikNet with calls to the C library's *malloc()* function; or 3) a collection of one or more blocks of memory acquired by KwikNet with calls to a user provided memory acquisition function *kn_msmemacquire()*.

If the KwikNet simple heap is dynamically constructed, its memory region will consist of one or more memory blocks called pages. For 32-bit systems, there is no particular advantage to splitting the memory region into multiple pages. However, for 16-bit segmented systems you may very well have to use multiple pages in order to provide a memory heap of more than 64K bytes. If the C library *malloc()* limits the amount of memory which it will allocate, you will have to provide your own memory acquisition function to provide the memory pages.

OS Memory Management

The third memory allocation strategy is to use the memory allocation services provided by the operating system (RT/OS) with which KwikNet is being used. The operating system may have its own private heap from which to allocate memory. However, some multitasking operating systems, such as the AMX kernel, permit the use of a separate memory region for privately allocating memory so that services such as those offered by KwikNet can operate without interference caused by the memory demands of other, unrelated tasks.

If the RT/OS requires its own memory region, it can be assigned in three ways: 1) a single static array in memory; 2) a single block of memory acquired by KwikNet with a call to the C library's *malloc()* function; or 3) a single block of memory acquired by KwikNet with a call to a user provided memory acquisition function *kn_msmemacquire()*.

Memory Acquisition Function

If you choose to do your own dynamic assignment of memory for a simple heap or for use by your RT/OS, then you must provide a memory acquisition function *kn_msmemacquire()* which KwikNet will call when it first starts. The prototype for this function is as follows:

```
unsigned long kn_msmemacquire(char **memp, unsigned long memsize);
```

The memory acquisition function must provide access to a fixed region of at least *memsize* bytes of memory. The function must install a pointer to the memory region into the pointer variable referenced as **memp* and return the value *n*, the actual size of the region provided by the function.

If you choose to use the KwikNet simple heap with multiple pages of memory, KwikNet will call your memory acquisition function *kn_msmemacquire()* repetitively to acquire each of the memory pages.

When KwikNet shuts down it will call a function *kn_msmemreturn()* which you must provide to dispose of the memory region, if necessary. The prototype for this function is as follows:

```
int kn_msmemreturn(char *memp);
```

Parameter *memp* is a pointer to the region of memory previously acquired from your *kn_msmemacquire()* function. If this function successfully handles the memory region disposal, it must return the value 0. Otherwise, it must return a non-zero value.

Memory Allocation Protection

When operating in a multitasking environment, the memory allocation services must be thread-safe. If the services you have chosen to use are not thread-safe, KwikNet can be configured to use the memory locking mechanism in its OS Interface Module to protect access to the unsafe memory allocation services.

If you use **standard C** for memory allocation within a multitasking system, you will have to use KwikNet's locking service to protect access to the library functions unless your C runtime library provides thread-safe implementations of its memory management functions.

If you use the KwikNet **simple heap** for memory allocation, the memory allocation services will be automatically locked by the Turbo Treck TCP/IP Stack if required. However, if you dynamically assign the memory page(s) in a multitasking system using the C library `malloc()` or your own memory acquisition function `kn_msmemacquire()`, you will have to use KwikNet's locking service to protect access to these functions unless they are inherently thread-safe.

If you use your **OS memory allocation** services in a multitasking system, you must be sure that they are thread-safe. If necessary, you can use KwikNet's locking service to protect access to the OS memory services. Special care must be taken if you dynamically assign a memory region for use by the OS. In this case, the C library function `malloc()` or your memory acquisition function `kn_msmemacquire()` will be called as soon as your OS is ready for use, long *before* KwikNet is operational. Hence you cannot use KwikNet's locking service to protect access to these functions. It is your responsibility to ensure that these functions are thread-safe at the time your application starts execution. In most cases this requirement will be met because the OS interface function `kn_osready()` executes before any other application tasks are started.

If KwikNet is used with AMX and standard C is used for memory allocation, you must enable the memory locking protection in the KwikNet OS interface to protect the unsafe services in the C library. If your C library is thread-safe, you will not require locking.

If the KwikNet simple heap is used with AMX, you will not require locking. The initial memory acquisition process and all subsequent memory allocation operations are automatically thread-safe.

If KwikNet is used with the AMX Memory Manager, you will not require locking. The initial memory acquisition process and all subsequent memory allocation operations are automatically thread-safe.

The KwikNet sample programs provided for use with AMX use the AMX Memory Manager to allocate memory from a static array. These services are thread-safe.

The examples provided with the KwikNet Porting Kit are ready for use with standard C. If you port KwikNet to a multitasking RTOS, be sure to implement the memory protection mechanism or provide access to thread-safe services within the RTOS.

1.6 KwikNet Data Logging Service

Like most TCP/IP stacks, KwikNet can generate a variety of messages to assist you in your use of the stack. The messages can provide debug information and trace execution of the stack through its various paths. Your application can also generate a statistics summary of event counts by calling KwikNet procedure *kn_netstats()*.

For debug and trace messages to be generated, the KwikNet Library must be configured accordingly. Statistics for a particular KwikNet component will only be available for logging if the KwikNet Library is appropriately configured. Even if the message sources are enabled, KwikNet will only log the messages if message logging is enabled. To enable these features, use the KwikNet Configuration Builder to edit your Network Parameter File as described in Chapter 2.3.

Edit the Network Parameter File as follows. Go to the Debug property page and check the box which enables message logging. On the same page, define the amount of memory you are willing to reserve for KwikNet to use for message storage. Specify the maximum allowable message line length, usually about 80 to 128 characters. Finally, enter the name of the data log function to be used by KwikNet to *print* each message. This function will be described shortly.

The KwikNet Library, ready for message logging use, will be generated and linked with your application as described in Chapter 3.

Message Formatting

Many TCP/IP stacks produce these data logging messages using the C library *printf()* function which is often not even available in embedded systems. KwikNet provides its own procedure called *kn_dprintf()* which, although similar to *printf()*, has several special features not found in the latter. This procedure is fully described in Chapter 4.6. The prototype is as follows.

```
int kn_dprintf(int attrib, const char *fmt, ...);
```

Unlike *printf()*, KwikNet's procedure receives a parameter *attrib* which defines the print attributes of the message. This encoded parameter defines the severity of the message, the message class and the message source. These characteristics will be described shortly.

The parameter *fmt* is a pointer to a conventional format string which can be followed by zero or more parameters as required by the format specification. Not all standard C formats are supported. For example, parameters of type *float* and *double* are not permitted. However, a new format "*%a*" is introduced which greatly simplifies the formatting of network IP addresses in dotted decimal notation.

For a complete specification of the formatting features supported by KwikNet procedure *kn_dprintf()*, see the description of format procedure *kn_fmt()* in Chapter 4.6.

Message Print Attributes

The parameter defining the message print attributes includes three fields of interest to the user: severity level, class and source type. These fields can be isolated using the following symbolic masks defined in KwikNet header file *KN_COMN.H*. All other bits in the parameter are reserved for the private use of KwikNet.

<i>KN_PA_LEVEL</i>	Severity level
<i>KN_PA_CLASS</i>	Message class
<i>KN_PA_TYPE</i>	Source type

The severity levels are defined as follows:

<i>KN_PA_INFO</i>	General information and application messages
<i>KN_PA_WARN</i>	KwikNet warnings
<i>KN_PA_FATAL</i>	KwikNet fatal error messages

The message classes, defined as follows, can be used to identify the device to which the messages should be directed.

<i>KN_PA_APP</i>	General information and application messages
<i>KN_PA_DEBUG</i>	KwikNet debug logging
<i>KN_PA_STATS</i>	KwikNet network statistics

The message source types define the module, network, device, protocol layer or service which was executing when the message was generated. The list is extensive and subject to change. The source types, defined in KwikNet header file *KN_COMN.H*, will generally be of little interest to your application. The source types could be used to provide a sub-classification if you wish to archive messages in some manner.

The message print attributes are defined such that an attribute of 0 will always describe an application message of lowest severity and with no known source type. Hence, applications can easily call *kn_dprintf()* with an attribute of 0 to log messages.

KwikNet Data Log Function

When message logging is enabled, the KwikNet message generation procedure *kn_dprintf()* calls the data log function specified in your Network Parameter File. It is the purpose of this function to record (and display or print) the message contained in the KwikNet log buffer which it receives.

The Application OS Interface module *KNSAMOS.C* provided for use with KwikNet sample programs includes a working example of a data log function called *sam_record()*. With some modifications, this procedure may be suitable for use by your application. At the very least, it will provide a good model for you to use.

The data log function must be declared as follows:

```
int sam_record(int attrib, char *bufp, int count);
```

The character buffer referenced by pointer *bufp* is a KwikNet log buffer. It contains a '\0' terminated string. The length of the string in bytes is specified by parameter *count*. Message strings are limited to the line length which you specified in your configuration. The newline character '\n' is used as the end of line indication in all KwikNet messages.

Parameter *attrib* defines the message print attributes. This is the same parameter presented to KwikNet's *kn_dprintf()* procedure. Your log function can decode the message class to determine the device on which the message must be recorded or displayed. It can also decide if any special action is required because of the message severity or source.

Finally, your log function must assume responsibility for the KwikNet log buffer. If your function accepts the log buffer, it must eventually release it by passing the pointer *bufp* to KwikNet procedure *kn_logbuffree()*. In this case, your log function must return the value 0 to KwikNet indicating your acceptance of the log buffer.

If your log function cannot accept the log buffer for some reason, it must return the value -1 to KwikNet. In this case, KwikNet will free the log buffer.

In a multitasking system, the log function should add the log buffer to a message queue for eventual recording (and printing or display) by a print task which services the message queue. The examples provided for use with AMX and with the KwikNet Porting Kit pass the log buffer to a print task which uses the KwikNet message recording service described in Chapter 1.7 to dispose of each message.

In a single threaded system, the log function should add the log buffer to a message queue for eventual recording (and printing or display) by the App-Task. However, if performance is not an issue, the log function can actually record the message and release the log buffer itself. Care must be taken to ensure that such an action is not allowed to occur while executing within the interrupt domain. The examples provided with the KwikNet Porting Kit operate in the latter fashion, using the KwikNet message recording service described in Chapter 1.7 to dispose of each message.

1.7 KwikNet Message Recording Service

Recognizing that embedded systems may not be able to display or print messages, KADAK provides an alternate message recording service. This service is provided in module *KNRECORD.C* which is located in the toolset dependent installation directory *TOOLXXX\SAM_COMN* (see Chapter 3.6).

The KwikNet message recording service, used by all KwikNet sample programs, accepts a message contained in a KwikNet log buffer. The message is copied from the log buffer into a memory array and the log buffer is released.

The messages are stored sequentially in a character array called *kn_records[]*. As each message is recorded, a pointer to the copy of the message is stored into the next available entry in variable *kn_recordlist[]*, an array of string pointers. The list of string pointers is terminated with a *NULL* string pointer. Message recording ceases as soon as either array becomes full.

Procedure *kn_loginit()* in module *KNRECORD.C* must be called by your application before the message recording service can be used by KwikNet. For this reason, your *main()* function should call *kn_loginit()* as one of its earliest operations.

Once the service is ready, procedure *kn_logmsg()* can be called to record a message contained in a KwikNet log buffer. The Application OS Interface module *KNSAMOS.C* used by KwikNet sample programs provides an example. The data log function *sam_record()* in that module ensures that each KwikNet log buffer is eventually delivered to procedure *kn_logmsg()* which records the message and releases the log buffer.

The data recording service can be adapted to your needs by editing the definitions in the sample program's application header file *KNZZZAPP.H*. A unique header file is provided with each KwikNet sample program. Symbol *KN_REC_MEMORY* must be set to 1 to enable recording of messages into character array *kn_records[]*. Symbol *KN_REC_MEMSIZE* defines the size of that array. Symbol *KN_REC_NUM* defines the maximum number of message string pointers which can be recorded into array *kn_recordlist[]*. If symbol *KN_REC_CONSOLE* is set to 1, each recorded message will also be echoed to the KwikNet console driver as described in Chapter 1.8.

Some of the KwikNet sample programs implement a dump command to display the recorded messages. These applications call procedure *kn_loggets()* to extract each message string from the recording array. After displaying all messages in the order in which they were recorded, procedure *kn_loginit()* is called to reset the array.

Also note that some debuggers will allow you to dump the strings in text form in a display window by viewing the array variable *kn_recordlist[]*.

Warning

The procedures in the recording module *KNRECORD.C* are NOT reentrant. Hence, in multitasking systems, you must ensure that, if one task calls any one of these procedures, no other task can execute any of the procedures until that task completes its use of the recording service.

1.8 KwikNet Console Driver

The KwikNet sample programs provide support for a simple, interactive console device. The console driver, module *KNCONSOL.C*, is located in the toolset dependent installation directory *TOOLXXX\SAM_COMN* (see Chapter 3.6). The console driver can be adapted to use any of several possible console devices, including a terminal connected by a serial UART interface, a PC screen and keyboard or a remote Telnet terminal.

To select a particular console device, edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* as instructed in the file. Note that a unique application header file *KNZZZAPP.H* is provided with each KwikNet sample program.

The basic KwikNet TCP/IP Sample Program uses the console device for displaying messages logged by KwikNet and the application. The data recording procedure *kn_logmsg()* in module *KNRECORD.C* echoes each message it receives to the console device. You can disable this display of recorded messages by setting the value of symbol *KN_REC_CONSOLE* to 0 in the sample program's application header file *KNZZZAPP.H*.

Other KwikNet sample programs (FTP Option, Web Server, etc) provide a simple command interpreter which allows you to interact with the program to control its operation. Since the console device is used by the application, it cannot be used by the recording service to display KwikNet messages. Hence, for these programs, symbol *KN_REC_CONSOLE* is defined to be 0 in the sample program's application header file *KNZZZAPP.H*.

The interactive KwikNet sample programs implement a dump command to display the recorded messages. These applications call procedure *kn_loggets()* in module *KNRECORD.C* to extract all of the message strings from the recording array. The extracted messages are displayed on the console device.

Warning

The message recording services are not reentrant. Hence, the dump command implemented by some KwikNet sample programs should only be used when KwikNet is not active since the extraction of messages for display may occur concurrently with the generation of messages by KwikNet.

If you use the Telnet console device, the dump command must be used with caution. Since KwikNet must be active for the Telnet console driver to operate, KwikNet may generate several messages for every message that is dumped, especially if you have enabled most of the KwikNet debug and trace options.

Serial I/O Terminal as the Console Device

The KwikNet sample program includes a UART serial I/O driver which can be used with the KwikNet console driver to provide access to a terminal. The driver supports the INS8250 (NS16550) USART as implemented in PC compatible hardware. To select this device for console driver use, edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVUART*. This serial I/O driver can also be used with the KwikNet sample programs for AMX.

The UART driver *KN8250S.C* is located in the toolset dependent installation directory *TOOLXXX\SAM_COMN* (see Chapter 3.6). Compile the console driver *KNCONSOL.C* and the UART driver *KN8250S.C* and link the resulting object modules with the sample program.

PC Display/Keyboard as the Console Device

When used on PC hardware with MS-DOS, the KwikNet console driver can be directed to use the PC display and keyboard as a terminal. Edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVPC*. The PC display and keyboard can only be used with a C library that supports the non-standard *_putch()*, *_kbhit()* and *_getch()* functions. This console device can also be used with the KwikNet sample programs for AMX 86.

Telnet as the Console Device

If the KwikNet sample program is modified to provide access to a real network, the KwikNet console driver can be directed to use the Telnet protocol to access a remote terminal. Edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVTELNET*. The KwikNet Library must have TCP support included. The console driver will listen on the well known Telnet port number 23 for a connection. It then uses the TCP socket to communicate with the remote terminal to which it is connected. The Telnet console device can also be used with the KwikNet sample programs for AMX.

AMX Console Devices

When using KwikNet with AMX, the KwikNet console driver can be used with the KwikNet serial UART driver described above. However, if you have already ported the AMX Sample Program serial I/O driver to your hardware, you can direct the console driver to use it to access a terminal. Edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVAMX*. Compile the console driver *KNCONSOL.C* and link the resulting object module and your AMX serial driver object module with the sample program.

If you are using AMX 86, the KwikNet console driver can use the AMX PC Supervisor to access the PC display and keyboard as a terminal. Edit the sample program's application header file *KNZZZAPP.H* and change the definition of symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVAMXPCS*. Be sure to link the sample program with the AMX PCS Configuration Module and the PC Supervisor Library.

1.9 Debugging Aids

KwikNet includes a number of debug features which, if used effectively, can help you test your networking application. These features can also be used to provide information to KADAK's technical support staff should you require their assistance.

KwikNet's debugging services fall into the following categories: debug logging, breakpoint traps and fatal error detection. Most of these features can only be used to best advantage if your application provides a data log function as described in Chapter 1.6.

Debug Logging

To use KwikNet's debug logging features, you must first build the KwikNet Library to include the extra code necessary to detect and record the events of interest. To do so, use the KwikNet Configuration Builder to edit your KwikNet Network Parameter File and view the Debug property page (see Chapter 2.3).

Check the box labeled "Informational messages". Doing so enables KwikNet to display messages which indicate the operations being performed by KwikNet. These messages provide a high level trace of the sequence of events as KwikNet executes.

If you check the box labeled "Error reporting", KwikNet will display warning messages if it detects error conditions which, although abnormal, do impair the ability of KwikNet to operate correctly. KwikNet will also display error messages if it detects serious error conditions which compromise the integrity of KwikNet. These messages indicate that all is not well with KwikNet.

Programmed Halt

If error reporting is enabled, you can check the box labeled "Halt on warnings/fatal errors". In this case, whenever KwikNet generates a warning message or fatal error message, it will immediately "stop" by spinning forever in the context of the task operating at the time the fault was detected. Interrupts will remain enabled so that higher priority tasks will still continue to execute in multitasking systems. But KwikNet will be stalled, unable to proceed.

Fatal Errors

Some of the errors detected by KwikNet are serious enough to require that KwikNet cease operation. To proceed would risk further corruption and would probably lead to a catastrophic collapse in an unpredictable fashion.

When KwikNet detects such a fatal error, it calls procedure *kn_panic()* which attempts to log a message describing the fault and then stop the RT/OS. When used with AMX, KwikNet forces an AMX fatal exit.

When testing, it is always wise to execute with a breakpoint on procedure *kn_panic()*. If you are using KwikNet with AMX, you should also have a breakpoint on the AMX fatal exit procedure *cjksfatal* (*ajfat1* and *AAFATL* for AMX 86).

Breakpoint Traps

KwikNet can generate a debug trap when it encounters an error condition which is generally not expected in the normal course of events. Such errors are often the result of modifications of private KwikNet data by errant applications which result in decision conflicts which KwikNet cannot resolve.

KwikNet debug traps are also generated whenever error reporting is enabled and a warning or error message is generated.

To use KwikNet's debug trap, you must first build the KwikNet Library to include the extra code necessary to generate the trap. To do so, use the KwikNet Configuration Builder to edit your KwikNet Network Parameter File and view the Debug property page (see Chapter 2.3). Check the box labeled "Trap to `kn_bphit()`".

Each debug trap generates a call to the KwikNet breakpoint procedure `kn_bphit()`. When testing your application, you can place a breakpoint on this procedure to trap all errors detected by KwikNet.

Note

KwikNet breakpoint procedure `kn_bphit()` will be called whenever a programmed halt occurs, even if the debug trap feature is not enabled. Hence you can always detect a programmed halt by placing a breakpoint on function `kn_bphit()`.

Monitoring Memory Usage

To use KwikNet's memory usage monitor, you must first build the KwikNet Library to include the extra code necessary to detect and record memory allocation and release. To do so, use the KwikNet Configuration Builder to edit your KwikNet Network Parameter File and view the Debug property page (see Chapter 2.3).

Check the box labeled "Monitor memory usage". Doing so enables KwikNet to monitor the total memory allocated for use by KwikNet at any instant. KwikNet also monitors the worst case memory usage. The memory usage summary can be displayed with other network statistics by calling KwikNet function `kn_netstats()`.

Debug Mask

KwikNet maintains a public debug control variable, an unsigned integer named *kn_dbgflags*. The bits in this variable are used to determine which, if any, KwikNet debug features are enabled at runtime. The bit masks *KN_DBxxxxx* used to access this variable are defined in KwikNet header file *KN_COMN.H*.

If bit *KN_DBENABLE* in variable *kn_dbgflags* is set to 0, all logging of error messages (warnings and fatal error) is disabled.

If bit *KN_DBENABLE* in variable *kn_dbgflags* is set to 1, debug message logging is enabled.

If logging is enabled and a warning or fatal message is logged (see Chapter 1.6), the action taken depends on the state of the *KN_DBHALT* bit in variable *kn_dbgflags*. If this bit is set to 1, KwikNet will generate a programmed halt, entering into a permanent loop, unconditionally calling its breakpoint procedure *kn_bphit()*.

If you have enabled a programmed halt, the debug halt bit *KN_DBHALT* will be set when your KwikNet application is loaded into memory. Otherwise, it will be clear. The halt bit is never altered by KwikNet.

You can dynamically alter bits *KN_DBENABLE* and *KN_DBHALT* in variable *kn_dbgflags* to enable and disable these debug features at runtime to best meet your debugging needs.

Note

Although the KwikNet variable *kn_dbgflags* is always present, its content will only have an effect if your application has been linked with a KwikNet Library which was built to include error reporting.

1.10 KwikNet TCP/IP Sample Program - A Tutorial

A TCP/IP Sample Program is provided with KwikNet to illustrate the use of the TCP/IP stack within an application. The sample program is ready for use with the AMX Real-Time Multitasking Kernel. The sample program has also been tested with each of the four porting examples provided with the KwikNet Porting Kit.

The sample configuration supports a single network interface. The network uses the KwikNet Ethernet Network Driver. Because the sample must operate on all supported target processors without any specific Ethernet device dependence, KwikNet's Ethernet Loopback Driver is used. Use of this driver provides two benefits: the illustration of a very simple device driver and an example of its use for testing purposes when network hardware is not available.

The KwikNet TCP/IP Stack requires a clock for proper network timing. The examples provided with the KwikNet Porting Kit all illustrate the clock interface. However, the sample program provided for use with AMX has been enhanced to eliminate any dependence on specific target hardware. This sample program includes a very low priority task which can detect if you have added a real AMX clock driver to the sample configuration. If a real hardware clock is not available, this task simulates clock interrupts, thereby providing AMX ticks which meet KwikNet's needs.

The sample includes two tasks, one acting as a server and the other as a client. In a multitasking system, these tasks are real tasks managed by the RTOS. In a single threaded system, the server is attached to the KwikNet Task's server queue and operates in the KwikNet domain. The client is simply the App-Task executing in the user domain.

The client and server use the KwikNet sockets application programming interface (API) to communicate. Two scenarios are followed, one after the other.

In the first scenario, the server creates a streaming socket, listens to the socket for a connection request, accepts a message from the client and generates the correct response. The client creates a streaming socket, establishes a connection with the server, sends its request and verifies the proper response.

In the second scenario, the server creates a UDP (connectionless) socket and listens for incoming requests. The server then uses the socket select feature to wait for the availability of a message from the client. The server reads the message, identifies the message source (client) and sends the correct response back to that client. The client creates a UDP socket, sends its request to the server, waits for the availability of a message from the server and verifies the proper response.

Once the final scenario has completed, the client calls KwikNet to log a summary of the network statistics accumulated during the session.

The sample uses the KwikNet message recording service (see Chapter 1.7) to record messages generated by the server, the client and KwikNet. Messages are stored as an array of strings in memory but can be easily echoed to a console terminal (see Chapter 1.8).

Startup

The manner in which the KwikNet TCP/IP Sample Program starts and operates is completely dependent upon the underlying operating system with which KwikNet is being used. All sample programs provided with KwikNet and its optional components share a common implementation methodology which is described in Appendix E. Both multitasking and single threaded operation are described in detail.

When used with AMX, the sample program operates as follows. AMX is launched from the *main()* program. Restart Procedure *rrproc()* starts a print task. A low priority background task is also started to simulate clock interrupts in the absence of a hardware clock. The Restart Procedure then calls application function *app_prep()* which creates and starts the client and server tasks.

Once the AMX initialization is complete, the high priority print task executes and waits for the arrival of AMX messages in its private mailbox. Each AMX message includes a pointer to a log buffer containing a KwikNet message to be recorded.

Once the print task is ready and waiting, the server task starts and waits for a signal from the client task. Then the client task finally begins to execute. It starts KwikNet at its entry point *kn_enter()*. KwikNet self starts and forces the KwikNet Task to execute. Because the KwikNet Task operates at a priority above all tasks which use its services, it temporarily preempts the client task. The KwikNet Task initializes the network and its associated loopback driver and prepares the IP and TCP protocol stacks for use by the sample program.

Once the KwikNet initialization is complete, the client resumes execution, signals the server to resume execution and begins the first of two scenarios.

Client - Server Using TCP Sockets

This example illustrates the use of KwikNet's TCP/IP socket interface to establish a connection between two end points for the reliable transfer of data. Although the end points happen to be tasks running on the same host computer, the actions required by each are still the same as would be required if they resided on separate hosts interconnected by a real network.

The server's first scenario is embodied in function *server1()*. The corresponding function executed by the client is *client1()*.

The **server** calls *kn_socket()* to create a streaming socket. The server calls *kn_setsockopt()* to revise the socket's attributes to permit the eventual reuse of the unique server port number of 5001. The server calls *kn_setsockopt()* again to force its socket to be non-blocking. It then calls *kn_bind()* to bind itself to the socket, identifying itself as port 5001, but allowing KwikNet to assign its IP address. KwikNet assigns IP address 192.168.1.73, the IP address of the only network present in the sample configuration. The server then calls *kn_listen()* to prime the socket to listen for incoming connection requests. Finally, the server calls *kn_accept()* to wait for such a connection to be established. KwikNet gives the server a new socket which corresponds to the server's end of the connection to its client.

Once the connection with the client has been made, the server uses procedure *kn_recv()* to receive a 4 byte message, a *long* value which defines the length of a block of data which the client intends to send to the server.

The server uses *kn_send()* to echo the 4 byte value back to the client as an acknowledgement that the server is prepared to accept that amount of data from the client. Then the server calls *kn_recv()* to acquire the data block from the client and *kn_send()* to echo the data block back to the client.

Once the data has been echoed to the client, the server uses *kn_shutdown()* to terminate all send operations on its connected socket and then repetitively calls *kn_recv()* until all unexpected data, if any, from the client has been discarded. The connected socket and the socket used for listening are then closed using *kn_close()*.

The **client** calls `kn_socket()` to create a streaming socket. It then calls `kn_bind()` to bind itself to the socket, allowing KwikNet to assign a port number and IP address.

The client then calls `kn_connect()` to connect to the server with port number 5001 and IP address 192.168.1.73. Note that the client has to know the port number of the server to which it is trying to connect. The IP address of the server happens to be the IP address assigned to the Ethernet network predefined in the KwikNet Library. The sample illustrates the use of procedure `kn_inet_addr()` to convert an IP address in dotted decimal form to an equivalent network compatible form.

Once the connection has been established, the client sends the *long* value 26 (defined as `SAMMSGSZ`) as a 4 byte message to the server. The number `SAMMSGSZ` is the number of data bytes which the client intends to send to the server. It happens to be the number of characters in the alphabet, the data which will be sent.

The client then uses procedure `kn_recv()` to receive a 4 byte message, a *long* value which confirms the length of the block of data which the server is willing to accept from the client.

The client then prepares to send the 26 characters of the alphabet to the server. The client does so using the `kn_writev()` procedure which permits the data to be gathered from disjoint locations in memory but delivered as a sequential byte stream. The 26 characters are gathered from the following two strings: 10 from the first and 16 from the second.

```
" ABCDEFGHIJ1234"  
" KLMNOPQRSTUVWXYZ5678"
```

The client then waits for an echo from the server of the data actually received by the server. The client does so using the `kn_readv()` procedure which permits the received data to be scattered into disjoint locations in memory even though received as a sequential byte stream. The data is scattered into a zero filled character buffer: 7 bytes at offset 0, 11 bytes at offset 20 and 8 bytes at offset 40. The three strings at offsets 0, 20 and 40 are then expected to match as follows.

```
" ABCDEFG"  
" HIJKLMNOPQR"  
" STUVWXYZ"
```

Finally, the client uses `kn_shutdown()` to terminate all send operations on its connected socket and then repetitively calls `kn_recv()` until all unexpected data, if any, from the server has been discarded. The socket is then closed using `kn_close()`.

Client - Server Using UDP Sockets

This example illustrates the use of KwikNet's UDP socket interface to deliver UDP datagrams between two end points. This data transfer mechanism is not considered reliable. Furthermore, since TCP is not used, a logical connection between the end points does not exist. Although the end points happen to be running on the same host computer, the actions required by each are still the same as would be required if they resided on separate hosts interconnected by a real network.

The server's second scenario is embodied in function *server2()*. The corresponding function executed by the client is *client2()*.

The **server** calls *kn_socket()* to create a connectionless datagram socket. The server calls *kn_setsockopt()* to force its socket to be non-blocking. It then calls *kn_bind()* to bind itself to the socket, identifying itself as port 5001, but allowing KwikNet to assign its IP address. KwikNet assigns IP address 192.168.1.73, the IP address of the only network present in the sample configuration. Once the bind is complete, the server can immediately receive data directed to IP address 192.168.1.73.

The server then uses procedure *kn_select()* to wait until some data from a client is available for reading. The sample illustrates the proper use of the macros *FD_ZERO*, *FD_SET* and *FD_ISSET* for manipulating socket sets.

Once data from the client is available, the server uses *kn_recvfrom()* to learn the client's network address and to receive a 4 byte message, a *long* value, from the client. The server then uses *kn_connect()* to identify the client to which it must send its response.

The server then uses procedure *kn_select()* to wait until the server's socket is ready to accept data for transmission. The value received from the client is incremented by two and echoed to the client using the *kn_send()* procedure to send the 4 byte *long* value.

Once the value has been echoed to the client, the server breaks its association with the client using *kn_connect()* to identify its peer as port 0 and IP address 0. The server then uses *kn_shutdown()* to terminate all send and receive operations on its socket. The socket is then closed using *kn_close()*.

The **client** calls *kn_socket()* to create a connectionless datagram socket. It then calls *kn_bind()* to bind itself to the socket, allowing KwikNet to assign a port number and IP address.

The client then uses procedure *kn_select()* to wait until the client's socket is ready to accept data for transmission. The arbitrary value 8 is sent to the server using the *kn_sendto()* procedure to send the 4 byte *long* value. The value is sent to the server with port number 5001 and IP address 192.168.1.73.

Note that the client has to know the port number of the server to which it is trying to connect. The IP address of the server happens to be the IP address assigned to the Ethernet network predefined in the KwikNet Library. The sample illustrates the use of procedure *kn_inet_addr()* to convert an IP address in dotted decimal form to an equivalent network compatible form.

The client then uses procedure *kn_select()* to wait until some data from the server is available for reading. The sample illustrates the proper use of the macros *FD_ZERO*, *FD_SET* and *FD_ISSET* for manipulating socket sets. The client then uses procedure *kn_recv()* to receive a 4 byte message, a *long* value from the server and confirms that the received value is 10, the value sent incremented by 2.

Once the echoed value has been received from the server, the client uses *kn_shutdown()* to terminate all send and receive operations on its socket. The socket is then closed using *kn_close()*.

Logging

The KwikNet TCP/IP Sample Program includes a simple recorder for logging text messages. The recorder saves the recorded text strings in a 30,000 byte memory buffer until either 500 strings have been recorded or the memory buffer capacity is reached.

The application can generate messages by calling the KwikNet log procedure *kn_dprintf()*. This procedure operates similarly to the C *printf()* function except that an extra integer parameter of value 0 must precede the format string. The sample program uses this feature to record startup and shutdown messages. The client and server record progress messages and log errors as they are detected.

KwikNet formats the message into a log buffer and passes the buffer to an application log function for printing. Log function *sam_record()* in the KwikNet Application OS Interface serves this purpose.

In a multitasking system the log buffer is delivered as part of an RTOS dependent message to a print task. The print task calls *kn_logmsg()* in the KwikNet message recording module to record the message and release the log buffer.

In a single threaded system, the log function *sam_record()* can usually call *kn_logmsg()* to record the message and release the log buffer. However, if the message is being generated while executing in the interrupt domain, the log buffer must be passed to the KwikNet Task to be logged. The sample programs provided with the KwikNet Porting Kit illustrate this process.

Shutdown

Once the client and server have completed their second scenario, the client calls KwikNet procedure *kn_netstats()* to record all network statistics gathered by KwikNet during the session.

The client then calls procedure *kn_exit()* to stop operation of the KwikNet TCP/IP Stack. A final completion message is then logged. Note that the KwikNet data logging services continue to be used by the application even though the stack itself has ceased operation. Finally, the client requests the operating system to shut down and return to the *main()* function.

Running the TCP/IP Sample Program

The KwikNet TCP/IP Sample Program load module is built just like any other KwikNet application program. The KwikNet Network Parameter File is first created using the KwikNet Configuration Builder as described in Chapter 2. Then the KwikNet Library must be produced. Finally the application must be linked with the KwikNet Library, the RT/OS libraries and the C runtime library to create a load module suitable for testing with a debugger. This construction process is explained in Chapter 3.

Since the KwikNet TCP/IP Sample Program has no visible output unless operated with a console terminal, its operation can only be confirmed using your debugger. Since the program has no hardware dependence, it can readily be used with a target processor simulator, if one is available.

KwikNet includes a number of debug features (see Chapter 1.9) which will assist you in running the TCP/IP Sample Program. With KwikNet's debug features enabled, you can place a breakpoint on procedure *kn_bphit()* to trap all errors detected by KwikNet. Of course, if you are using AMX, it is always wise to execute with a breakpoint on the AMX fatal exit procedure *cjksfatal* (*ajfat1* for AMX 86).

If you breakpoint at the end of the *main()* program, you can examine the messages recorded in memory. The messages are stored sequentially in a character array called *kn_records[]*. Variable *kn_recordlist[]* is an array of string pointers referencing the individual recorded messages. Most debuggers will allow you to dump the strings referenced in *kn_recordlist[]* in text form in a display window. The list of string pointers is terminated with a *NULL* string pointer.

If you are connected to the target processor by a serial link, do not be surprised if the debugger takes quite some time to access and display all of the strings referenced by *kn_recordlist[]*. You may be able to improve the response by limiting the display to the actual number of strings in the array as defined by variable *kn_recordindex*.

Once you are confident that the KwikNet TCP/IP Sample Program is operating properly, you may wish to breakpoint your way through the client and server (functions *clientN()* and *serverN()*), monitoring the recorded messages as you go.

2. KwikNet System Configuration

2.1 Introduction

Creating an application which uses the KwikNet TCP/IP Stack is a two step process. First, the KwikNet Library must be constructed to reflect the options and features which your application will require. If you wish, you can predefine one or more of the networks and devices present in your target hardware. Finally, your application modules must be linked with the KwikNet Library to create a load module suitable for execution in the target processor.

With many portable network stacks, this process requires the tedious and error prone task of editing a collection of files with which you have little familiarity. With KwikNet you simply point and click using the KwikNet Configuration Builder, a Windows[®] utility which greatly simplifies the process. You still have to pick the correct set of options and define your particular network requirements but at least you are concentrating on what you know best, your application.

KwikNet Library

The KwikNet Library must be constructed to reflect the options and features which your application will require. This information is kept in a text file called the KwikNet Network Parameter File which is created and edited for you by the KwikNet Configuration Builder. This editing process is illustrated in Figure 2.1-1.

From the KwikNet Network Parameter File, say *NETCFG.UP*, the Builder will generate a make specification file, a text file which can be used to create (make) the KwikNet Library as described in Chapter 3.2. This file, called the KwikNet Library Make File *NETCFG.MAK*, is created by merging information from the Network Parameter File with the Library Make Template File *KNnnnLIB.MT*.

The Builder can also generate a copy of the KwikNet Library Header File *KN_LIB.H* which is needed to compile KwikNet source files. This header file will also be included by any application module which uses KwikNet services. File *KN_LIB.H* is created by merging information from the Network Parameter File with the Library Header Template File *KNnnnLIB.HT*.

Note

If you use the KwikNet Library Make File *NETCFG.MAK* to create your KwikNet Library, you do not have to generate the KwikNet Library Header File *KN_LIB.H*. It will be created for you during the make process.

2.2 KwikNet Configuration Builder

The KwikNet Configuration Builder is a software generation tool which can be used to help create your KwikNet Library. You can think of the Builder as a very specialized editor. The Builder consists of two components: the Configuration Manager and the Configuration Generator. The Configuration Manager is an interactive utility which allows you to create and edit your Network Parameter File.

For convenience, the Configuration Manager has the ability to directly invoke its own copy of the Configuration Generator. The Configuration Generator reads your Network Parameter File and merges the information from it with a template file to produce an output text file. This process has been described in Chapter 2.1.

The Configuration Generator is also available as a separate, stand alone utility program that can be used within your make files to create, from your Network Parameter File and the KwikNet Template Files, any of the output text files which you usually use the Configuration Builder to generate.

Starting the Builder

The KwikNet Configuration Builder will operate on a PC or compatible running the Microsoft® Windows® operating system.

The KwikNet Configuration Builder is delivered with the following files.

File	Purpose
<i>KNnnnnCM .EXE</i>	KwikNet Configuration Manager (utility program)
<i>KNnnnnCM .CNT</i>	KwikNet Configuration Manager Help Content File
<i>KNnnnnCM .HLP</i>	KwikNet Configuration Manager Help File
<i>KNnnnnCG .EXE</i>	KwikNet Configuration Generator (utility program)
<i>KNnnnnLIB.MT</i>	KwikNet Library Make Template File
<i>KNnnnnLIB.HT</i>	KwikNet Library Header Template File

When KwikNet is installed on your hard disk, the KwikNet Configuration Manager utility program and its related files are stored in directory *CFGBLDW* in your KwikNet installation directory. To start the Configuration Manager, double click on its filename, *KNnnnnCM.EXE*. Alternatively, you can create a Windows shortcut to the Manager's filename and then simply double click the shortcut's icon.

Screen Layout

Figure 2.2-1 illustrates the Configuration Manager's screen layout. The title bar identifies the parameter file being created or edited. Below the title bar is the menu bar from which the operations you wish the Manager to perform can be selected.

Below the menu bar is an optional Toolbar with buttons for many of the most frequently used menu commands. The Toolbar is hidden or made visible using the View Toolbar command on the Edit menu.

The leftmost Toolbar button is used to create a new Network Parameter File.

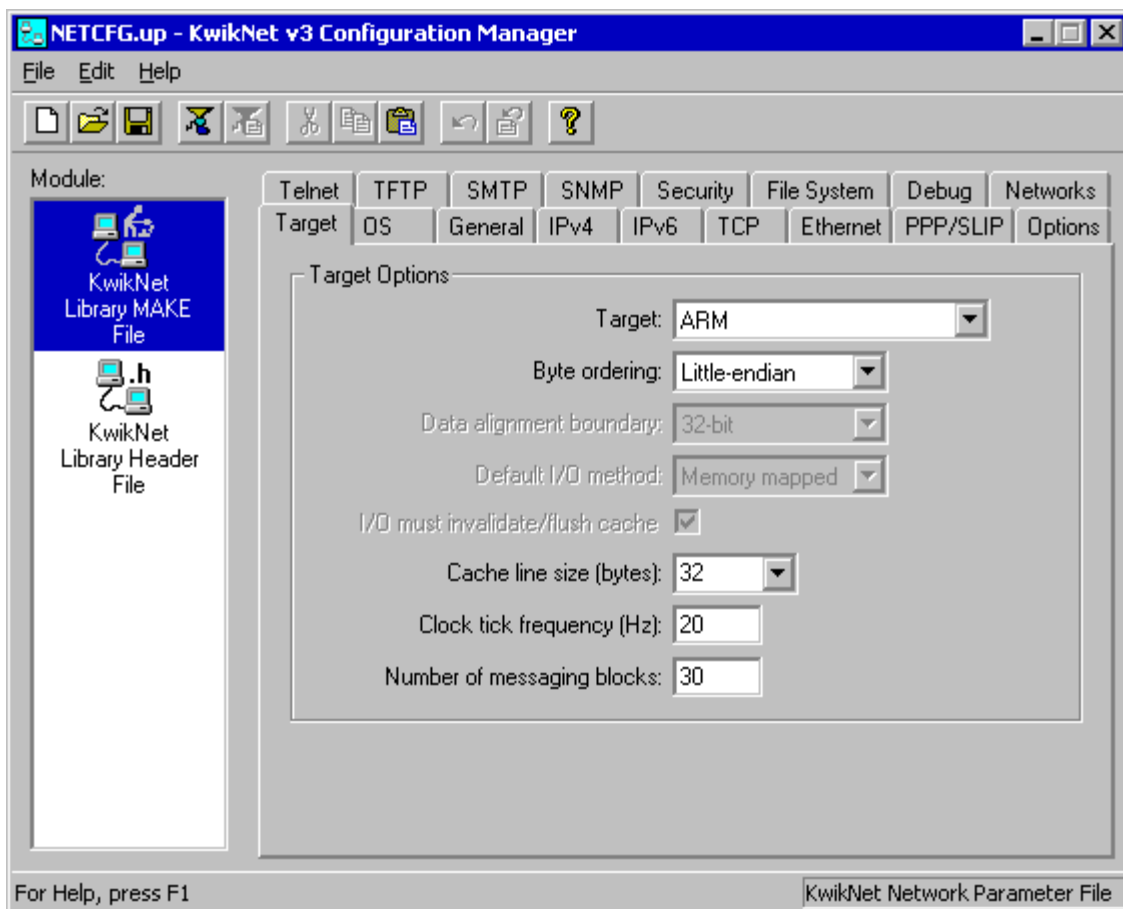


Figure 2.2-1 Configuration Manager Screen Layout

At the bottom of the screen is the status bar. As you select menu items, a brief description of their purpose is displayed in the status bar. If the Configuration Manager encounters an error condition, it presents an error message on the status bar describing the problem and, in many cases, the recommended solution.

Along the left margin of the screen are a set of one or more selector icons. These icons identify the type of output files which the Manager can produce from the parameter file being edited. The example illustrates that when editing a KwikNet Network Parameter File, the selectors for the KwikNet Library Header File and the KwikNet Library MAKE File are visible.

The center of the screen is used as an interactive viewing window through which you can view and modify your KwikNet network configuration parameters.

Menus

All commands to the Configuration Manager are available as items on the menus present on the menu bar. The **File menu** provides the conventional New, Open, Save and Save As... commands for creating and editing your parameter file. It also provides the Exit command.

Once a particular selector icon has been chosen as the currently active selector, the Generate... command on the File menu can be used to generate the corresponding output product. The path to the template file required by the generator to create this product can be defined using the Templates... command on the File menu.

The **Edit menu** provides the conventional Cut, Copy, Paste and Undo editing commands. It also includes an Undo Page command to restore the content of all fields on a property page to undo a series of unwanted edits to the page. The Toolbar is hidden or made visible using the View Toolbar command on the Edit menu.

The **Help menu** provides access to the complete KwikNet Configuration Manager reference manual. Context sensitive help is also available by pressing the F1 function key or clicking the ? button on the Toolbar.

Field Editing

When editing a parameter file, a collection of tabbed property sheets is displayed in the central region of the screen. Each tab provides access to a particular property page through which your configuration parameters can be declared. For instance, if while editing your Network Parameter File, you select the Networks tab, the Configuration Manager will present a network definition window (property page) containing all of the parameters you must provide to completely define a network.

Some fields are boolean options in which all you can do is turn the option on or off by checking or unchecking the associated check box.

Some fields are option fields in which you must select one of a set of options identified with radio buttons or pull down lists. Click on the option button or pick the list item which meets your preference.

Other fields may require numeric or text entry. Parameters are entered or edited in these fields by typing new values or text to replace the current field content. Only displayable characters can be entered. New characters which you enter are inserted at the current cursor position in the field. Right and left arrow, backspace and delete keys may be used to edit the field.

When you are altering a numeric or text field, you tell the Configuration Manager that you are finished editing the field by striking the Enter key. At that point, the Configuration Manager checks the numeric value or text string that you have entered for correctness in the context of the current field. If the value or text string that you have entered is invalid, an error indication is provided on the status bar at the bottom of the screen suggesting how the fault should be corrected.

The Tab and Shift-Tab keys can also be used to complete the editing of a field and move to the next or previous field.

If you have modified some of the fields on a property page and then decide that these modified values are not correct, use the Undo Page command on the Edit menu or Toolbar to force the Configuration Manager to restore the content of all fields on the page to the values which were in effect when you moved to that property page.

When you go to save your parameter file or prepare to move to another property page, the Configuration Manager will validate all parameters on the page which you are leaving. If any parameters are incomplete or inconsistent with each other, you will be forced to fix the problem before being allowed to proceed.

Add, Edit and Delete KwikNet Objects

Separate property pages are provided to allow your definition of one or more KwikNet objects such as networks and device drivers.

Pages of this type include a list box at the left side of the property page in which the currently defined objects are listed. At the bottom of the list box there may be a counter showing the number of objects in the list and the allowable maximum number of such objects.

Also below the list are two control buttons labeled Add and Delete. If the allowable maximum number of objects is 0 or if all such objects have already been defined, the Add button will be disabled. If there are no objects defined, the Delete button and all other fields on the page will be disabled.

To add a new object, click on the Add button. A new object with a default identifier will appear at the bottom of the list and will be opened ready for editing. When you enter a valid identifier for the object, your identifier will replace the default in the object list.

To edit an existing object's definition, double click on the object's identifier in the object list. The current values of all of that object's parameters will appear in the property page and the object will be opened ready for editing.

To delete an existing object, click on the object's identifier in the object list. Then click on the Delete button. Be careful because you cannot undo an object deletion.

The objects in the object list can be rearranged by dragging an object's identifier to the desired position in the list. You cannot drag an object directly to the end of the list. To do so, first drag the object to precede the last object on the list. Then drag the last object on the list to precede its predecessor on the list.

This page left blank intentionally.

2.3 KwikNet Library Configuration

The KwikNet Library tabbed property sheet is displayed in the central region of the screen. Each tab provides access to a particular property page through which your library configuration parameters can be declared. For instance, if you select the IPv4 tab, the Configuration Manager will present an IPv4 definition window (property page) containing all of the parameters you can adjust to completely define your use of the IPv4 protocol.

To create a new Network Parameter File, select New from the File menu. The Configuration Manager will create a new, as yet unnamed, file using its default KwikNet library configuration parameters. When you have finished defining or editing your library configuration, select Save As... from the File menu. The Configuration Manager will save your Network Parameter File in the location which you identify using the filename which you provide.

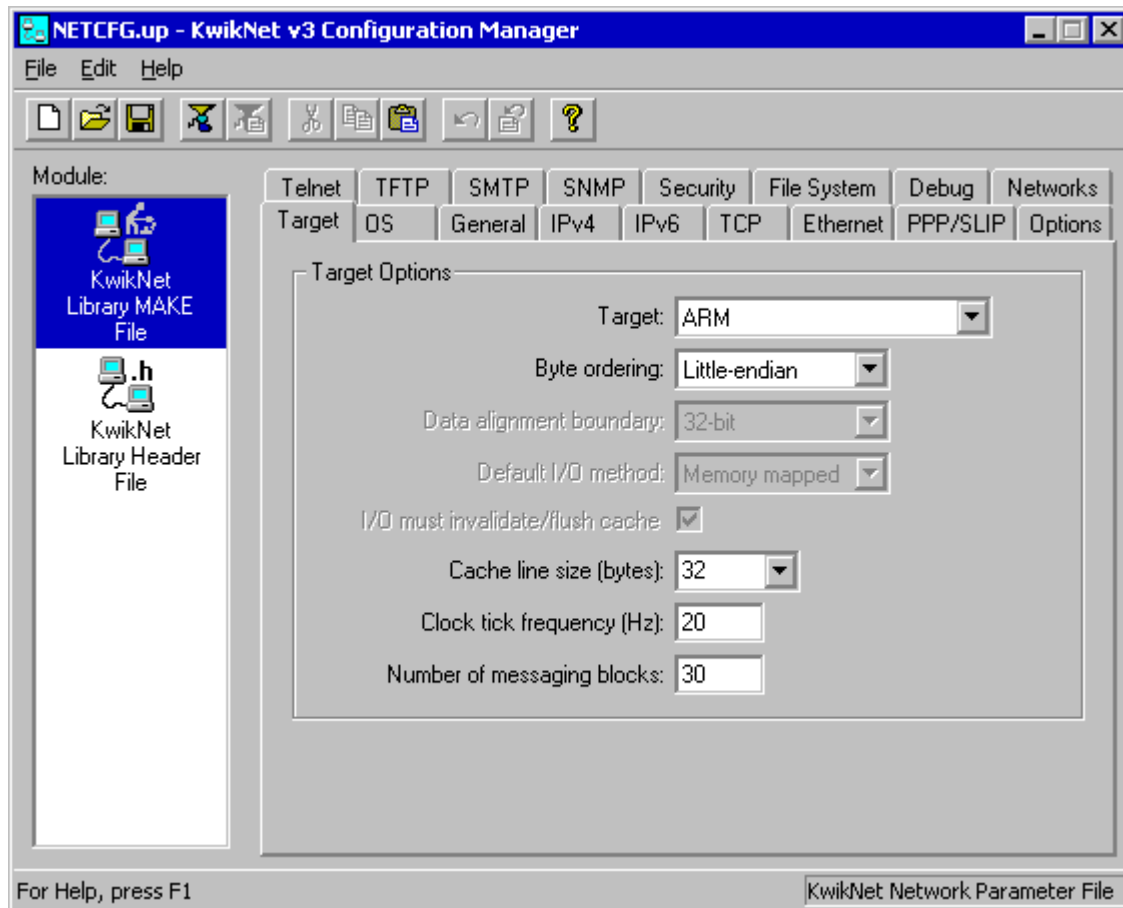
To open an existing Network Parameter File, say *NETCFG.UP*, select Open... from the File menu and enter the file's name and location or browse to find the file. When you have finished defining or editing your library configuration, select Save from the File menu. The Configuration Manager will rename your original Network Parameter File to be *NETCFG.BAK* and create an updated version of the file called *NETCFG.UP*.

When the KwikNet Library MAKE File selector icon is the currently active selector, the Generate... command on the File menu can be used to generate your Library Make File, say *NETCFG.MAK*. The path to the template file required by the generator to create this product can be defined using the Templates... command on the File menu.

When the KwikNet Library Header File selector icon is the currently active selector, the Generate... command on the File menu can be used to generate your Library Header File *KN_LIB.H*. The path to the template file required by the generator to create this product can be defined using the Templates... command on the File menu.

Target Parameters

The KwikNet Library must be tailored to operate on a particular target processor. These KwikNet parameters are edited using the Target property page. The layout of the window is shown below.



Target Processor

Select the target processor of interest from those available on the pull down list.

Byte Ordering

From the pull down list, choose the byte ordering scheme (big endian or little endian) used by the target processor's memory system. If the byte ordering method is dictated by the processor, this field will be preset and unalterable.

Target Parameters (continued)

Data Alignment Boundary

From the pull down list, choose the target processor's natural data alignment (16-bit or 32-bit) for long variables and structures. If the natural data alignment is dictated by the processor, this field will be preset and unalterable.

Default I/O Method and I/O Cache

From the pull down list, choose the method (memory mapped or I/O ports) used for device I/O addressing for the target processor. If the I/O addressing method is dictated by the processor, this field will be preset and unalterable.

When memory mapped I/O is used on processors like the PowerPC, it may be necessary to invalidate the data cache before I/O reads and flush the data cache after I/O writes. For such systems, check the I/O...cache box.

Cache Line Size

If your target processor has a data cache, then specify the size of a single line in the data cache. The most common values for this parameter can be selected from the pull down list. If your processor's data cache line size is not in the list, you must type the value directly into the edit field. If your target processor does not have a data cache, then set this field to 0.

KwikNet Clock Tick Frequency

Enter the frequency of the fundamental KwikNet clock tick. All KwikNet timing measurements will be based on this frequency. The KwikNet Task will perform its stack polling operations at this frequency.

The KwikNet clock frequency must be at least 2 Hz. A frequency of 10 Hz or 20 Hz is recommended. Any frequency much above 50 Hz will simply introduce unnecessary execution overhead with little noticeable improvement in network throughput.

Note that KwikNet must achieve the specified clock frequency using timing services provided by the underlying operating system. You must therefore choose a KwikNet clock frequency which is derivable by that operating system. Set the KwikNet clock frequency so that the corresponding period is an integral number of OS system ticks.

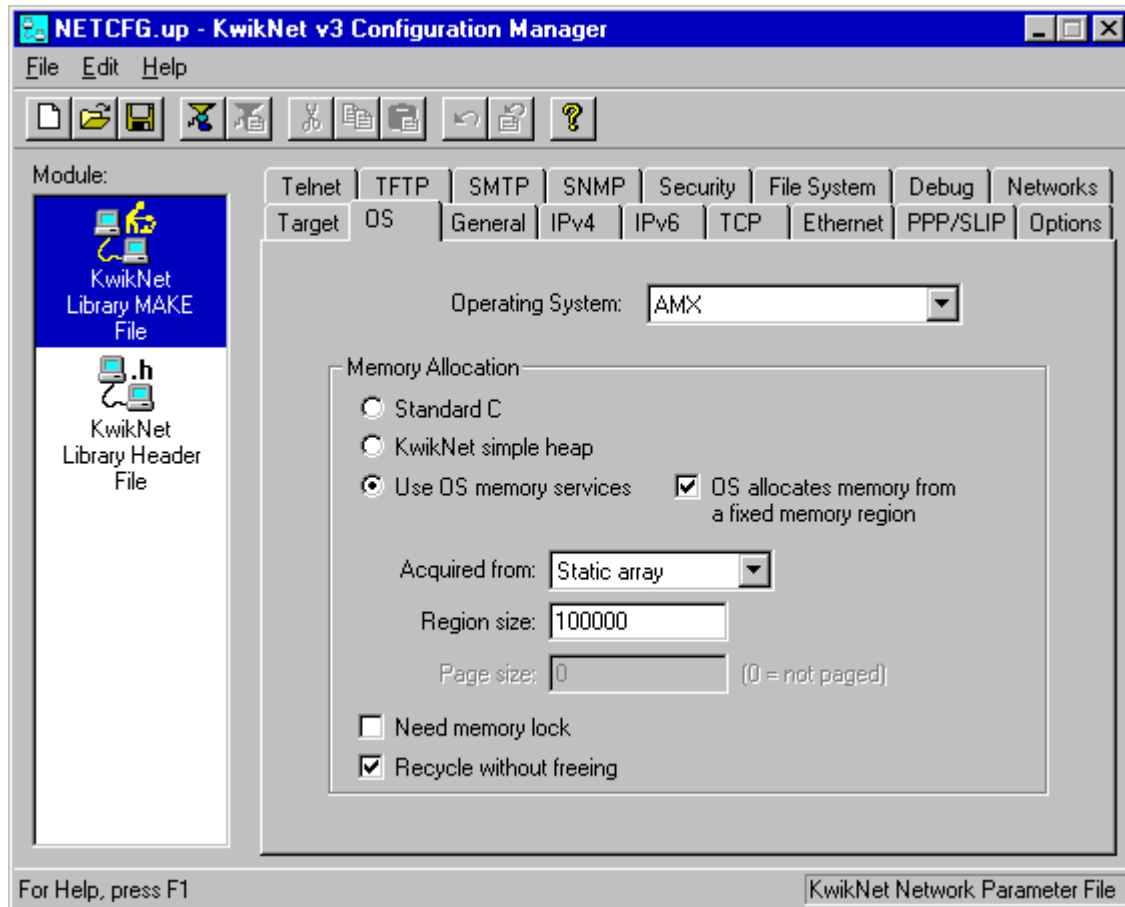
Number of Messaging Blocks

KwikNet uses messaging blocks for its private internal communication. You may have to increase the number of messaging blocks if any of the following conditions exist.

- You have many tasks using network services
- You service several networks concurrently
- You expect high levels of network packet activity

OS Parameters

The KwikNet Library must be tailored to operate with a particular operating system. These KwikNet parameters are edited using the OS property page. The layout of the window is shown below.



Operating System

From the pull down list, choose the underlying operating system upon which KwikNet must rely. KwikNet is ready for use without modification with KADAK's AMX kernels. Choose AMX for any of the 32-bit AMX kernels. Choose AMX 86 for 16-bit, real mode operation on any 80x86/88 processor.

The KwikNet Porting Kit can be used to port KwikNet to your operating environment, with or without an RTOS. The kit includes examples of KwikNet which work with the following single threaded operating systems: MS-DOS and the Tenberry DOS/4GW DOS Extender for MS-DOS. Custom implementations for a user defined in-house or commercial RTOS and for a single threaded OS are also provided with the kit.

OS Parameters (continued)

Memory Allocation

KwikNet must be able to dynamically allocate and free blocks of memory of varying sizes. Three methods of memory allocation are supported (see Chapter 1.5).

Standard C: If you check the "Standard C" radio button, KwikNet will use standard C library functions *malloc()* and *free()* to allocate and free memory.

Simple Heap: If you check the "KwikNet simple heap" radio button, KwikNet will provide its own simple heap and the services to use the heap to allocate and free memory. You must then choose the memory source and indicate its size and paging requirements by filling in the memory region parameters.

OS Services: If you are using an operating system which provides its own memory allocation services, check the radio button labeled "Use OS memory services".

If you are porting KwikNet to your own operating system and wish to use its memory allocation services, you must edit the functions which access those services in the OS Interface Module *KN_OSIF.C* (see Chapter 2 of the KwikNet Porting Kit User's Guide).

OS Fixed Memory Region (Heap)

When using the operating system's memory allocation services, you may have to provide memory for use as a heap. If so, check the box labeled "OS allocates memory from a fixed region". For example, when AMX memory management services are used, a private memory pool is created for network memory allocation, isolating KwikNet from conflicts arising from memory allocation for other uses. If you check this box, you must choose the memory source and indicate its size by filling in the memory region parameters.

OS Parameters (continued)

Source and Size of Memory Region (Heap)

If you use the KwikNet simple heap or operating system memory services which require a private region of memory for use as a heap, then you must provide that memory. From the pull down list, choose the method to be used to allocate such a memory region for use by the memory allocator.

If you choose the **Static array** option, you must enter the array size (in bytes) in the Region size field. A character array of that size will be declared in the KwikNet Library. KwikNet will allocate memory from that array.

If you choose the **malloc()** option, use the Region size field to define the number of bytes to be allocated for use as a heap. During KwikNet's startup initialization, the KwikNet Task will call C library function *malloc()* once to allocate a memory block of the specified size. KwikNet will subsequently allocate memory from that single block.

An alternate approach is to choose the **User function** option and provide a function called *kn_msmemacquire()* which is prototyped as shown below. The Region size field is passed to your function as parameter *memsize*. When KwikNet starts up, it will call the function to get the size of your memory region and a pointer to it. The function must install a pointer to a block of *n* bytes of memory into **mempp* and return the value *n* where *n* is greater than or equal to *memsize*.

```
unsigned long kn_msmemacquire(char **mempp, unsigned long memsize);
```

Protect Memory Get/Free Operations

When operating in a multitasking environment, the memory allocation services must be thread-safe. If the memory allocation services you have chosen to use are not thread-safe, check the box labeled "Need memory lock" and KwikNet will use its memory locking mechanism to protect access to memory. Otherwise, leave this box unchecked.

If you are using the AMX kernel and either the KwikNet simple heap or the AMX Memory Manager with a private heap, then memory allocation services are inherently thread-safe. Hence, leave this box unchecked.

If you are using KwikNet with AMX and using standard C for memory allocation you must check this box, *unless* your C library is inherently thread-safe.

When operating in a single threaded environment, memory allocation services are inherently thread-safe. Hence, leave this box unchecked.

OS Parameters (continued)

Recycle Without Freeing Memory

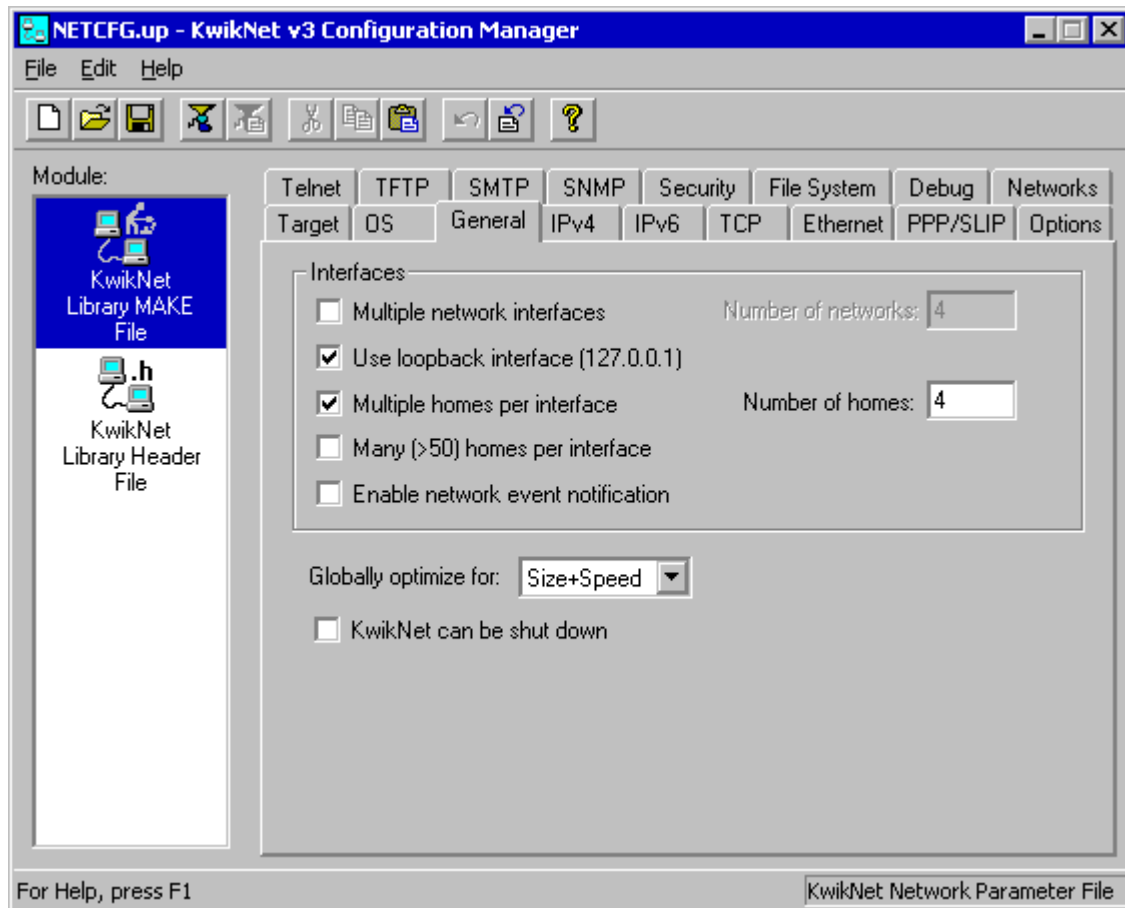
KwikNet can recycle memory which it has allocated instead of freeing the memory and having to re-allocate an equivalent sized block of memory at some future time. By recycling allocated memory blocks, KwikNet can improve performance once it reaches a steady state since the need for costly memory allocations is greatly reduced.

Of course, there is a penalty to be paid for this feature. And that is the code space needed for the recycling services and the extra data storage needed to manage the recycle bins.

To enable memory recycling, check the box labeled "Recycle without freeing". Otherwise, leave this box unchecked.

General Stack Parameters

A number of general network parameters which apply to the complete KwikNet TCP/IP stack are edited using the General property page. The layout of the window is shown below.



General Stack Parameters (continued)

Multiple Networks

Leave this box unchecked if you have only one network interface through which you can interconnect.

Check this box if your application must support more than one network interface. Such configurations are called multi-homed. Enter the maximum number of network interfaces which KwikNet must be able to support.

Use Loopback Interface (127.0.0.1)

If you have only one network interface, you can still use the loopback feature to permit packets destined for loopback IP address 127.0.0.1 to be looped back to KwikNet instead of being passed on to the device driver for transmission on the network. Check this box if you require loopback support with a single network interface. Otherwise, leave this box unchecked to reduce the memory footprint.

Note: Do not confuse this parameter with the KwikNet loopback device driver. The loopback driver simply accepts a packet for transmission and returns it to KwikNet, just as though the packet had been received from the network interface.

Multiple Homes Per Interface

KwikNet allows each network interface to be assigned more than one IP address. Such an interface is said to be multi-homed. Using this feature, even a system with only one network interface can be multi-homed.

If any of your network interfaces is to be multi-homed, check this box and enter the maximum number of homes which any one network interface is permitted to have. Otherwise, leave this box unchecked.

Many (>50) Homes Per Interface

If you intend to assign a large number of IP addresses (>50) to a multi-homed network interface, you can optimize the multi-homed IP address management services by checking this box. The resulting speed improvement comes with the penalty of increased code and data storage requirements. To reduce the memory footprint, leave this box unchecked.

Network Event Notification

KwikNet will notify your application whenever a significant event occurs on any network. Events include changes in network state (up, down, in-transit) and acquisition or loss of dynamically assigned IP addresses. To signal the event, KwikNet calls a function named *kn_netevent()* which you must provide. This application function is documented in Chapter 4.6.

If you wish to be notified when significant network events occur, check this box. Otherwise, leave this box unchecked.

General Stack Parameters (continued)

KwikNet Optimization

The KwikNet Library can be optimized for speed or size. In general, speed improvement comes at the expense of increased code size. Conversely, reducing the memory footprint usually imposes an execution speed penalty. You can choose your optimization preference from the pull down list.

KwikNet Shutdown

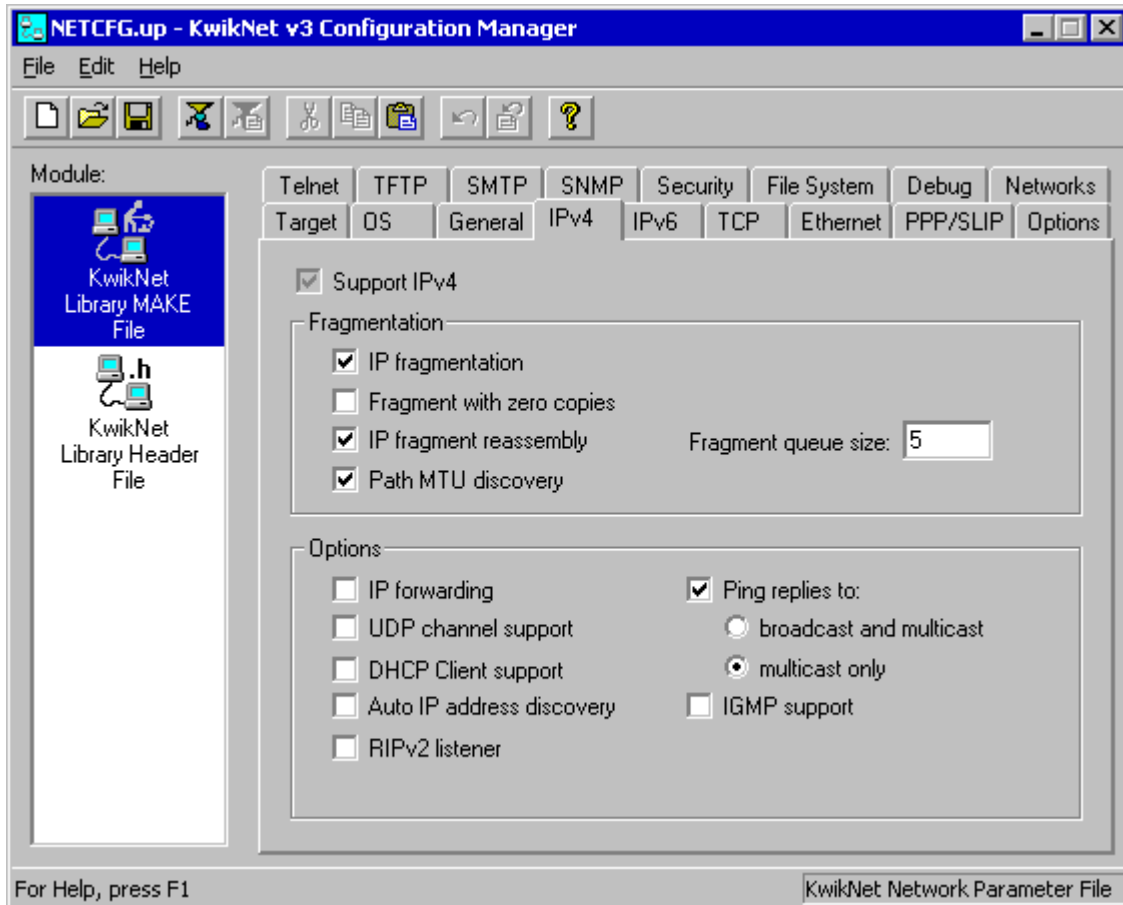
Most embedded applications which incorporate a network stack operate forever once started. In some cases however, especially when testing, it may be desirable to permit the network stack to shut down in an orderly fashion. If you plan to call KwikNet function *kn_exit()* to shut down KwikNet, check this box. Otherwise, leave this box unchecked and all of the shutdown code will be omitted, thereby reducing the memory footprint.

Warning!

You must not shut down KwikNet unless KwikNet's memory is allocated from a static array or from a fixed region of memory acquired when KwikNet is started and released when KwikNet is stopped.

IPv4 Stack Parameters

The KwikNet IPv4 Stack parameters are edited using the IPv4 property page. The layout of the window is shown below.



IPv4 Stack Parameters (continued)

IP Fragmentation

If you wish KwikNet to be able to split IP datagrams for transmission via a network interface or path with a reduced IP data capacity, check this box. Otherwise, leave this box unchecked.

Fragment with Zero Copies

If you are using custom device drivers which interface directly with the Turbo Treck TCP/IP Stack and support its scattered send mechanism, then you can avoid an extra data copy when an IP datagram has to be fragmented for transmission via your network interfaces. To do so, check this box. Otherwise, leave this box unchecked.

If you are using any standard KwikNet device driver or a device driver derived from the KwikNet Ethernet or serial shell driver, leave this box unchecked.

IP Fragment Reassembly

If you wish KwikNet to be able to reassemble IP datagrams which arrive as multiple IP datagram fragments, check this box and enter the maximum number of datagram fragments which KwikNet must be able to queue for reassembly. Otherwise, leave this box unchecked.

Path MTU Discovery

In order to determine if an IP datagram must be fragmented for transmission on a specific network interface, KwikNet must determine the Maximum Transmission Unit (MTU) (largest IP data payload) which the path to a particular destination permits. If you wish KwikNet to use the Path MTU Discovery technique specified in RFC1191 to determine the MTU to be used for communication with a particular destination, check this box. Otherwise, leave this box unchecked to reduce the memory footprint.

IP Forwarding

If you want to support packet forwarding between IPv4 interfaces, check this box. Otherwise, leave this box unchecked.

Note that packets will only be forwarded between IPv4 interfaces that are configured to support forwarding.

IPv4 Stack Parameters (continued)

UDP Channel Support

KwikNet offers a simplified UDP service for those not familiar with the use of the UDP sockets API. The KwikNet UDP service uses a UDP channel, an artificial connection for sending and receiving UDP datagrams (see Chapter 4.1). If you intend to use KwikNet UDP channel services, check this box. Otherwise, leave this box unchecked.

DHCP Client Support

KwikNet includes a DHCP Client for using the Dynamic Host Configuration Protocol (DHCP) to derive an IP address, subnet mask and default gateway for any of your network interfaces. The DHCP Client also supports the BOOTP protocol. The DHCP Client is described in Chapter 4.2. The DHCP Client is provided as a standard feature with systems that only incorporate IPv4. For dual IPv4/IPv6 systems requiring a DHCP Client, an alternate, optional DHCP component is available.

If any network interface must use DHCP or BOOTP for IP address acquisition, check this box. Otherwise, leave this box unchecked.

Auto IP Address Discovery

The optional KwikNet Auto IP component can be used to dynamically establish an IP address for any of your Ethernet network interfaces. If any Ethernet network interface must use Auto IP for IP address negotiation, check this box. Otherwise, leave this box unchecked.

RIP v2 Listener

KwikNet supports the Routing Information Protocol (RIP). If you check this box, KwikNet will listen for RIP packets and add the routing information which they provide to the KwikNet routing table. If you do not require RIP support, leave this box unchecked.

PING Replies To

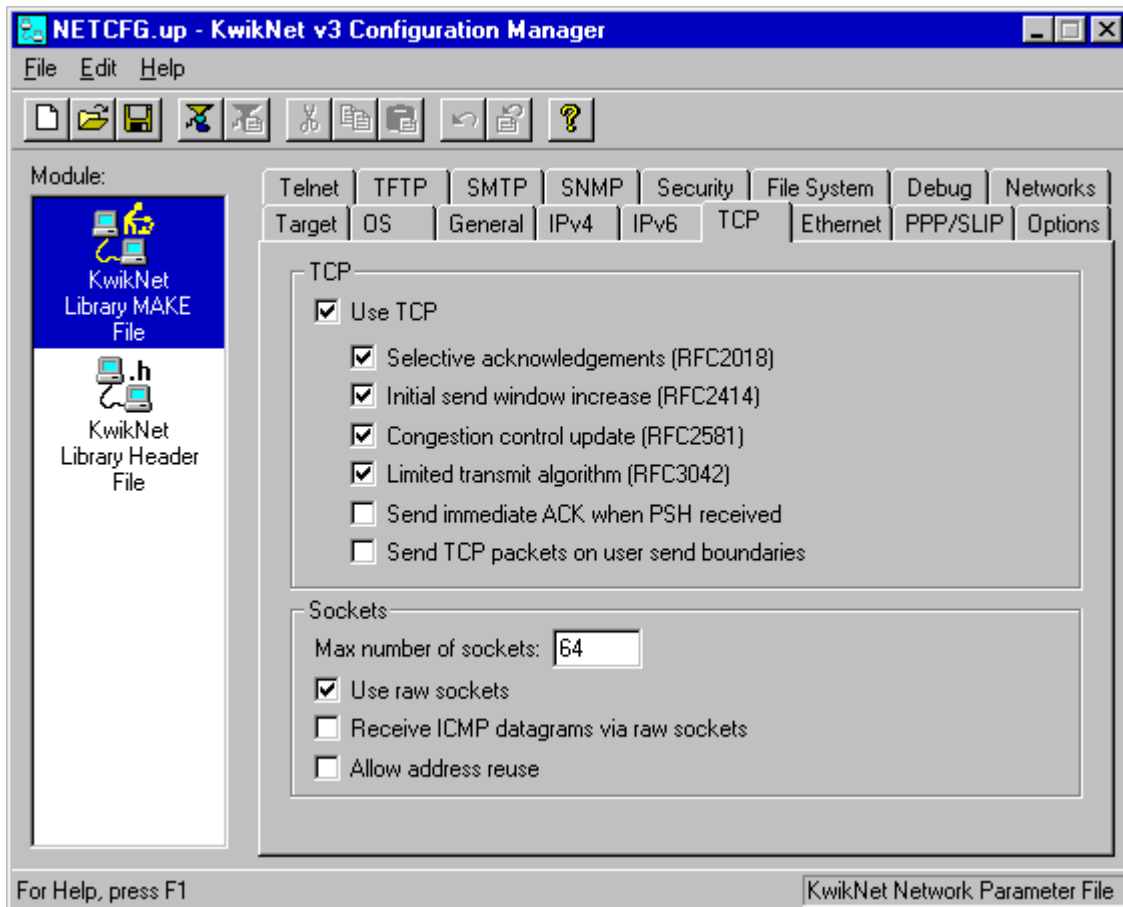
Check this box if you wish KwikNet to respond to an ICMP ping request addressed to a broadcast or multicast IP address. Check one of the radio buttons to indicate whether KwikNet should respond to "broadcast and multicast" or to "multicast only". If all ping requests to a broadcast or multicast IP address are to be ignored, leave this box unchecked.

IGMP Support

Check this box if you have the optional KwikNet IGMP component and wish to include it in the KwikNet Library.

TCP Stack Parameters

The KwikNet TCP Stack parameters and socket interface options are edited using the TCP property page. The layout of the window is shown below.



Include TCP

Check this box if you intend to use the TCP protocol and its KwikNet socket interface. Otherwise, leave this box unchecked to reduce the memory footprint.

TCP Stack Parameters (continued)

RFC2018: Selective Acknowledgements

This box is normally checked so that KwikNet will follow the selective acknowledgment strategy as documented in RFC2018. Leave this box unchecked to disable this feature.

RFC2414: Initial Send Window Increase

This box is normally checked so that KwikNet will increase the initial TCP send window as recommended in RFC2414. Leave this box unchecked to disable this feature.

RFC2581: Congestion Control

This box is normally checked so that KwikNet will follow RFC2581 which updates the recommended TCP congestion control strategy documented in RFC2001. Leave this box unchecked to disable this feature and ignore the update.

RFC3042: Limited Transmit Algorithm

This box is normally checked so that KwikNet will follow the limited transmit algorithm documented in RFC3042. The algorithm is intended to enhance the TCP Loss Recovery process. Leave this box unchecked to disable this feature.

Send Immediate ACK

Check this box if you want KwikNet to immediately send an *ACK* response whenever it receives a TCP packet with the *PSH* flag set, requiring immediate delivery of the data to the application. This action may improve performance when connected to a TCP peer running Windows 2000. It is usually best to leave this box unchecked to disable this feature.

Packets on User Boundaries

Check this box if you want TCP data packet transmission to be tightly coupled to your application's requests to send data. Each request to send data will force a new TCP packet to be generated. Performance will be sacrificed but the application will be able to control the manner in which its data is presented in packets on the network. Leave this box unchecked to disable this feature for best performance.

Note: If this feature is used, Path MTU Discovery (see IPv4 Parameters page) and TCP Selective Acknowledgements (see above) will be disabled!

TCP Stack Parameters (continued)

Maximum Number of Sockets

Enter the maximum number of sockets which your application can have in use at any one time. Remember that a TCP server needs one socket to listen for connection requests and one socket for each accepted connection. Also note that sockets are required for use with UDP or KwikNet UDP channels. Sockets are also used privately by many components of Turbo Treck TCP/IP Stack.

Use Raw Sockets

Check this box if you wish the sockets API to support raw sockets so that your application can send and received IP datagrams. Otherwise, leave this box unchecked.

Note: The KwikNet sockets API (see Chapter 5) does not support raw sockets. You must use the Treck raw sockets API.

ICMP via Raw Sockets

Check this box if you wish the raw socket API to support the delivery of IP datagrams for ICMP echo requests (ICMP type 8) and address mask requests (ICMP type 17) to raw sockets. Otherwise, leave this box unchecked.

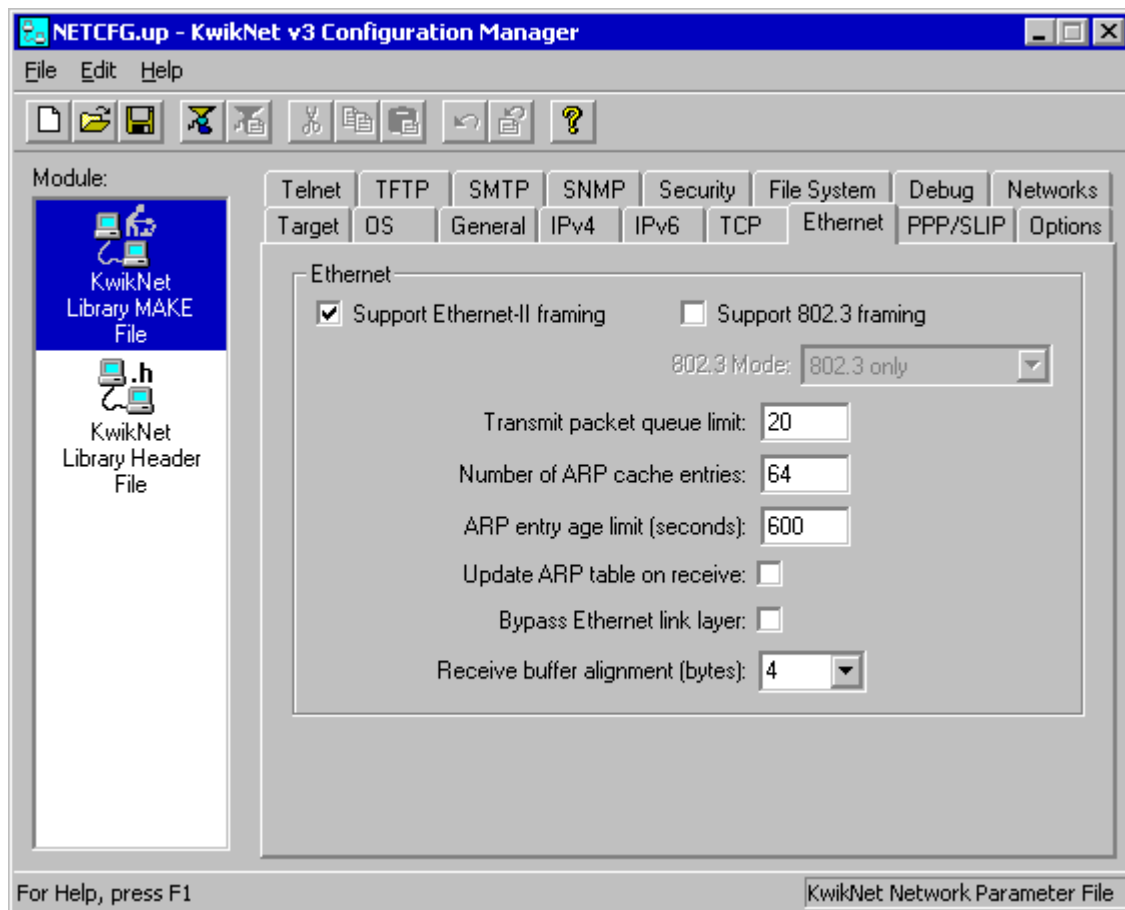
Allow Address Reuse

Check this box if you want to be able to bind multiple sockets with different IP addresses to the same local port. Otherwise, leave this box unchecked.

This feature must be enabled in order to set the `SO_REUSEADDR` socket option with a call to function `kn_setsockopt()`.

Ethernet Parameters

KwikNet includes an Ethernet network driver which can support multiple Ethernet networks. From the Ethernet property page, you can select the Ethernet network driver for inclusion in the KwikNet Library and edit its operating parameters. The layout of the window is shown below.



Support Ethernet

If you have one or more networks with Ethernet device drivers, check one or both of the Ethernet framing choices to include the KwikNet Ethernet Network Driver in your KwikNet Library. Otherwise, leave both boxes unchecked.

Ethernet Parameters (continued)

Ethernet-II Framing

Check this box if you have one or more network interfaces that only support Ethernet-II (DIX) framing. If none of your network interfaces use Ethernet-II (DIX) framing exclusively, leave this box unchecked.

This option must be enabled if you plan to create an Ethernet-II network, either dynamically at runtime or by predefining it on the Networks property page. If you create an Ethernet-II network and this option is not enabled, your application's link will fail because function *kn_etnet_prep()* is not defined.

802.3 Framing

Check this box if you have one or more network interfaces that must support Ethernet 802.3 framing. If none of your network interfaces use 802.3 framing, leave this box unchecked.

Note: Support for Ethernet 802.3 framing will not be present (even if you check this box) unless your Treck installation includes the optional Ethernet 802.3 component.

This option must be enabled if you plan to create an Ethernet 802.3 network, either dynamically at runtime or by predefining it on the Networks property page. If you create an Ethernet 802.3 network and this option is not enabled, your application's link will fail because function *kn_et8023_prep()* is not defined.

802.3 Operating Mode

If any Ethernet network interface requires 802.3 framing, you must select the mode of operation from the pull down list. All network interfaces that use 802.3 framing will operate according to the mode that you select.

Select **802.3 only** if these interfaces use 802.3 framing exclusively.

Select **802.3 (some DIX)** if these interfaces mainly use 802.3 framing but must occasionally use Ethernet-II (DIX) framing.

Select **DIX (some 802.3)** if these interfaces mainly use Ethernet-II (DIX) framing but must occasionally use 802.3 framing.

Ethernet Parameters (continued)

Transmit Packet Queue Limit

The KwikNet Ethernet Network Driver queues packets for transmission until they can be accepted for transmission by the Ethernet device driver. This parameter defines the maximum number of packets which the network driver will queue.

Number of ARP Cache Entries

The KwikNet Ethernet Network Driver uses the Address Resolution Protocol (ARP) to associate a specific hardware Ethernet address with a particular IP address. This parameter defines the maximum number of ARP address pairs which KwikNet can maintain in its ARP data cache for all supported networks.

If your network interfaces are interconnected with only a few other hosts, set this parameter to the number of such interconnections.

ARP Entry Age Limit

This parameter defines the length of time, measured in seconds, that an ARP address pair can reside in the ARP cache before it is purged. Any ARP entry whose age exceeds this time limit will not be used for address resolution.

Update ARP Table on Receive

Check this box if you want KwikNet to update its ARP table every time it receives a packet. By doing so, KwikNet can avoid having to periodically send ARP requests for foreign IP addresses that are being actively used for communication. However, there will be a performance penalty, since every received packet will be checked for an ARP cache match. Leave this box unchecked for best performance and a reduced memory footprint, albeit with more ARP requests than might otherwise be necessary.

Bypass Ethernet Link Layer

If this box is checked, KwikNet will bypass some of its more general link layer processing if the network interface supports Ethernet. Doing so can improve performance with a very minor increase in code size. If you are using an Ethernet interface and can afford the memory penalty, check this box for best performance. Otherwise, leave this box unchecked.

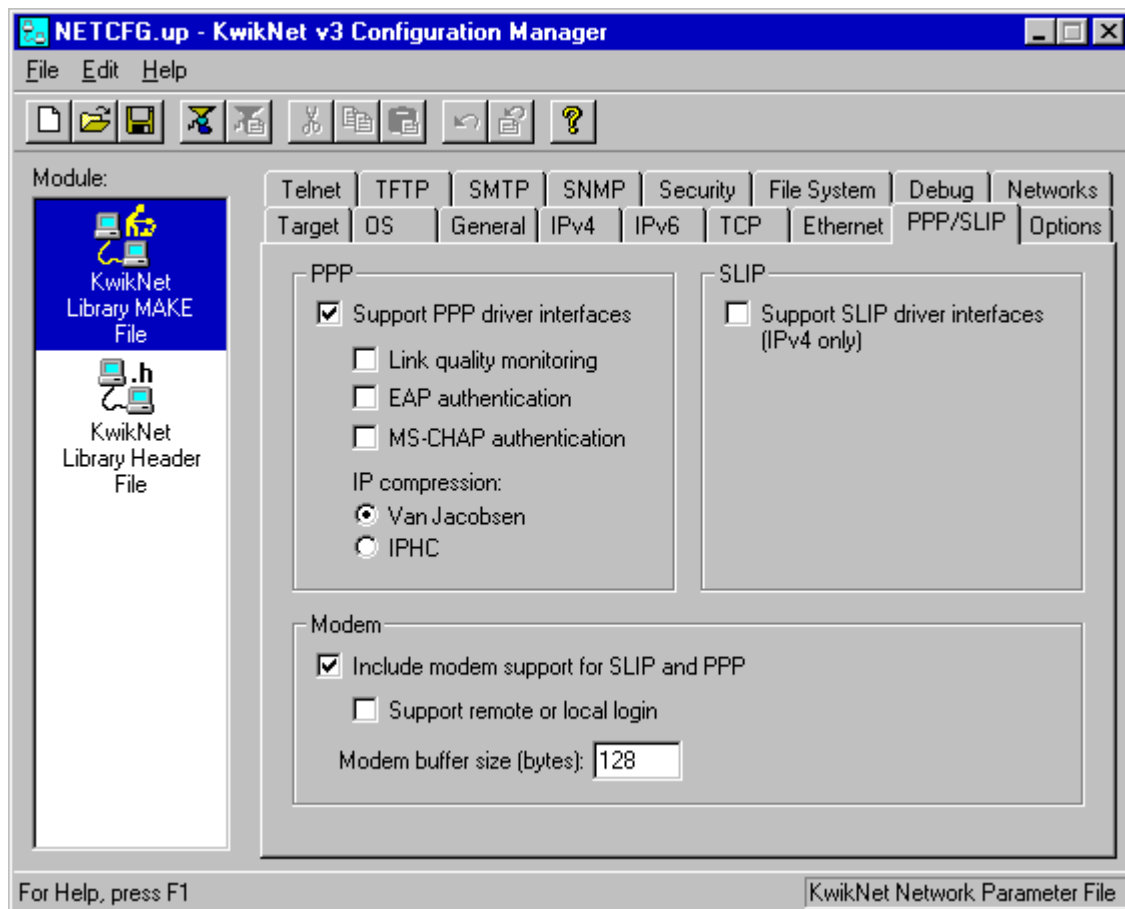
Receive Buffer Alignment

This parameter defines the memory alignment of all KwikNet packet buffers, whether used for transmitting or receiving Ethernet packets. This alignment permits some Ethernet devices with DMA engines to write received data directly to the KwikNet packet buffer.

If you have multiple Ethernet devices which will benefit from such alignment, enter the worst case (maximum) receive buffer alignment required by the devices.

Point-to-Point (PPP) / SLIP / Modem Parameters

KwikNet has an optional Point-to-Point (PPP) network driver which is capable of supporting multiple PPP networks. The standard KwikNet product includes a SLIP network driver which is capable of supporting multiple SLIP networks. From the PPP/SLIP property page, you can select either of these network drivers for inclusion in the KwikNet Library. This property page is also used to edit the PPP operating parameters. The layout of the window is shown below.



Support SLIP

If you have one or more SLIP networks with serial UART device drivers, check this box to force inclusion of the SLIP Network Driver in the KwikNet Library. Otherwise, leave this box unchecked.

PPP / SLIP / Modem Parameters (continued)

Support PPP

If you have one or more PPP networks with serial UART device drivers, check this box to force inclusion of the PPP Network Driver in the KwikNet Library. Otherwise, leave this box unchecked.

Link Quality Monitoring

Check this box to enable use of the Link Quality Protocol for monitoring the quality of the serial communication link. Otherwise, leave this box unchecked.

IP Header Compression

The KwikNet PPP option supports two forms of IP header compression which can improve transfer times over serial links if large numbers of small packets are being transferred. Van Jacobson (VJ) Compression is a technique for compressing TCP and IP headers. RFC2507 describes the IP Header Compression (IPHC) protocol, an algorithm which is not restricted for use with TCP datagrams.

Check one of the two radio buttons to select the IP header compression algorithm to be used with PPP networks which permit IP header compression. The Van Jacobson algorithm has the smaller memory footprint.

EAP Authentication

Check this box if any of your PPP networks must support the Extensible Authentication Protocol (EAP) to negotiate authorized use of the serial link. Otherwise, leave this box unchecked.

MS-CHAP Authentication

Check this box if any of your PPP networks must support the Microsoft Challenge-Handshake Authentication Protocol (MS-CHAP) to negotiate authorized use of the serial link. Otherwise, leave this box unchecked.

PPP / SLIP / Modem Parameters (continued)

Include Modem Support

KwikNet includes a modem driver for use with any SLIP or PPP network driver. The modem driver can operate concurrently on multiple networks.

If you have one or more SLIP or PPP networks with serial UART device drivers requiring modem support, check this box to include the KwikNet Modem Driver in your KwikNet Library. Otherwise, leave this box unchecked.

Support Remote or Local Login

The KwikNet Modem Driver can support local or remote login scripts. If this box is checked, then a local login script can be used by the modem driver to control the dialing and login sequence when a local user attempts to login to a remote server. A remote login script can be used to control the answering and login sequence when a remote user attempts to login to a local server.

Leave this box unchecked if login scripting is not required on any network serviced by a modem.

The KwikNet Modem Driver is described in Chapter 1.7 of the KwikNet Device Driver Technical Reference Manual.

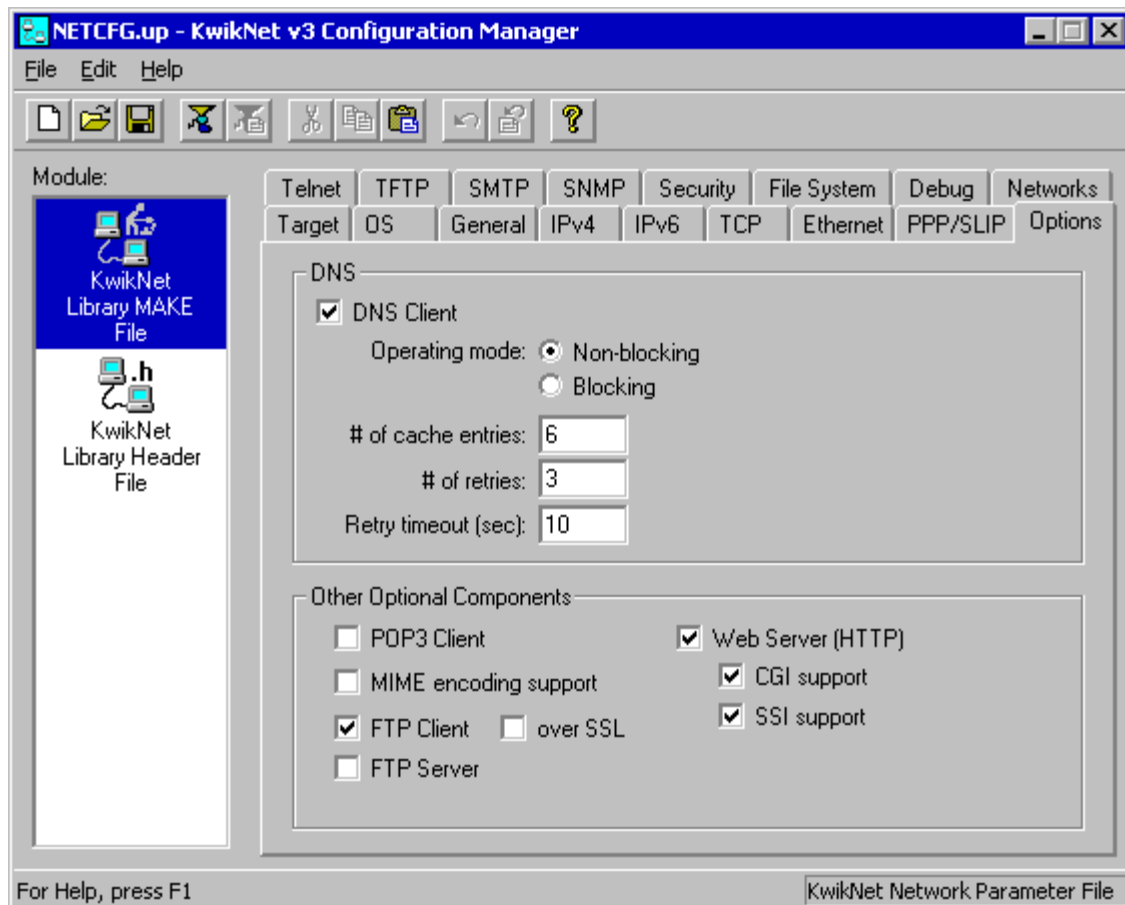
Modem Buffer Size (Bytes)

This parameter defines the size of the modem buffers used by the modem driver to send modem commands and receive modem responses when configuring the modem. If login scripting is enabled, the buffers are also used for sending or receiving login text strings. The buffer size must be large enough to hold the longest modem configuration string, modem response string or login string, including the ' \0 ' string termination character.

DNS Client Parameters

KwikNet includes an optional DNS Client for accessing Domain Name System services on interconnected networks. The DNS Client is described in Chapter 4.3. The DNS Client is selected and its parameters edited using the Options property page.

The layout of the Options window is shown below.



DNS Client Parameters (continued)

Include DNS Client

If your application needs access to Domain Name System services on interconnected networks, check this box. Otherwise, leave this box unchecked.

Operating Mode

The KwikNet DNS Client will operate in either non-blocking mode or blocking mode. Check the radio button to select the mode in which your application expects to operate when using DNS services.

Number of Cache Entries

The KwikNet DNS Client maintains a list of the domain names which your application has queried. This parameter defines the maximum number of domain names which you wish the DNS Client to cache. The larger you make this parameter, the more memory will be allocated for caching.

Number of Retries

This parameter defines the number of extra attempts which the DNS Client will make to acquire the IP address for a particular domain name before declaring failure.

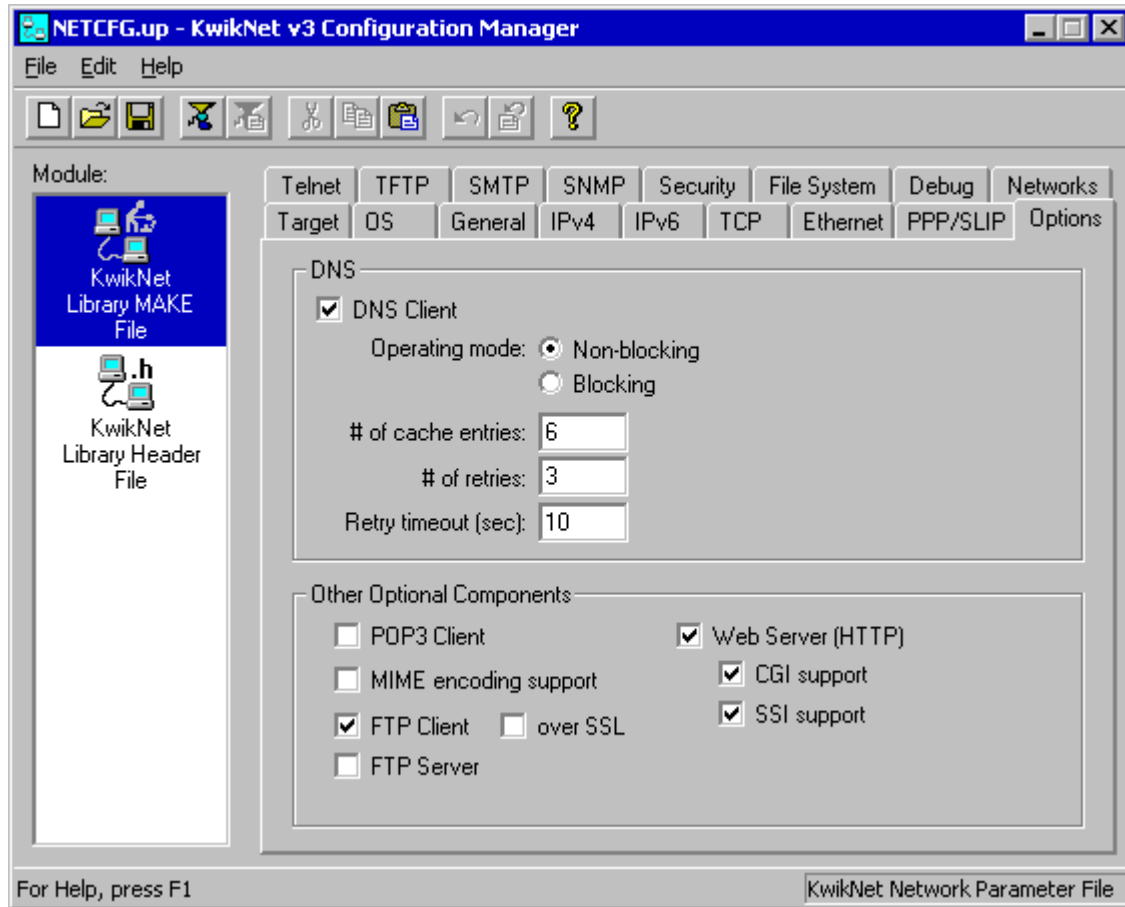
Retry Timeout

This parameter defines the number of seconds which the DNS Client will wait for a response to a DNS query. If no response is received within this timeout period, the DNS Client resends its query. This process repeats up to the maximum number of retries which you have specified.

Optional Components

KwikNet includes a variety of client and server options, many of which are selected using the Options property page.

The layout of the Options window is shown below.



POP3 Client

Check this box if your application will include a POP3 client which connects to a POP3 server for mail retrieval operations. Otherwise, leave this box unchecked.

MIME Encoding Support

Check this box if your application uses MIME encoding and decoding features. Otherwise, leave this box unchecked.

Optional Components (continued)

FTP Client

Check this box if your application will include an FTP client which connects to an FTP server for file transfers. Otherwise, leave this box unchecked.

Check the box labeled "over SSL" if any FTP client will use the Secure Sockets Layer (SSL) for secure file transfers. Otherwise, leave the box unchecked.

Note: To use SSL with the FTP client, you require the optional version of the FTP client which includes SSL support.

Note: If you choose the FTP client over SSL option, any FTP client can operate with or without security, according to the requirements of each particular FTP session.

FTP Server

Check this box if your application will include an FTP server to provide network access to files present in your target system. Otherwise, leave this box unchecked.

Web Server (HTTP)

Check this box if your application will include a web server to provide network access to your target system. Otherwise, leave this box unchecked.

CGI Support

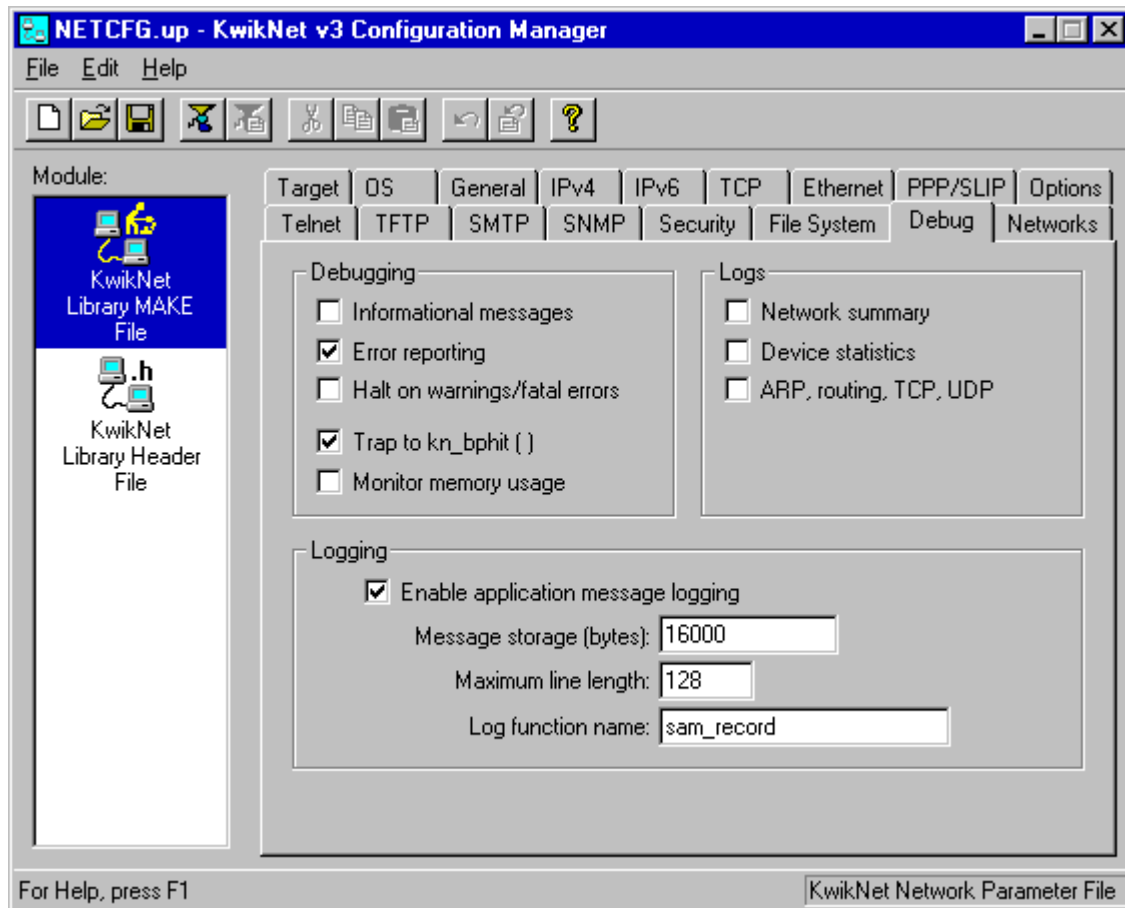
Check this box if your HTTP web server application will include support for dynamically generated web pages using the Common Gateway Interface (CGI). Otherwise, leave this box unchecked.

SSI Support

Check this box if your HTTP web server application will include support for web pages with dynamically generated embedded content produced using Server Side Include (SSI) technology. Otherwise, leave this box unchecked. CGI support must be enabled before SSI support can be enabled.

Debug and Logging Parameters

KwikNet includes a number of debugging and data logging options. These options are enabled and selected using the Debug property page. The layout of the window is shown below.



Debug and Logging Parameters (continued)

Informational Messages

Check this box to enable informational messages to be generated to help monitor the progress of KwikNet as it executes. If this box is checked, code is inserted into the KwikNet Library to generate progress messages. To omit this code, leave this box unchecked.

Error Reporting

Check this box to enable warning and error messages to be generated to help identify unexpected problems encountered by KwikNet as it executes. If this box is checked, code is inserted into the KwikNet Library to generate warning and error messages. To omit this code, leave this box unchecked.

Halt on Warnings and Fatal Errors

Check this box to force KwikNet to halt if it generates a warning or fatal error message. KwikNet will attempt to display the message on the recording device and will then spin forever in a compute bound loop calling the breakpoint trap function *kn_bphit()*.

If you do not want KwikNet to stop in this fashion, leave this box unchecked.

Error Trap

Many of the procedures in the KwikNet TCP/IP Stack and its optional modules can generate a debug trap after logging a debug message when an error is detected. Check this box if you wish all such debug traps to be vectored to the KwikNet error trap procedure *kn_bphit()* on which you can place a debugger breakpoint. Otherwise, leave this box unchecked.

Monitor Memory Usage

Check this box to monitor memory usage. KwikNet will monitor all memory allocated by functions *kn_msalloc()* or *kn_msgetzero()* and freed by function *kn_msfree()*. The results can be viewed using function *kn_netstats()*. To omit this feature and the code used to monitor memory usage, leave this box unchecked.

Debug and Logging Parameters (continued)

Enable Message Logging

Check this box if you wish to provide a logging function to record (display) text messages generated by KwikNet's data logging services. When this box is checked, you must provide all of the logging parameters listed. If you do not want to provide a logging function, leave this box unchecked.

Message Storage

Specify the amount of memory to be reserved for use as log buffers. A character array of this size will be declared in the KwikNet Library.

Maximum Line Length

Specify the maximum number of characters that are allowed in each line of text logged to your recording (display) device.

Log Function Name

This parameter provides the name of an application function which will be called to log each line of text generated by KwikNet's data logging services. This function must operate as specified in Chapter 1.6.

The sample programs provided with KwikNet and its optional components use data logging procedure *sam_record()* in the Application OS Interface module *KNSAMOS.C*. To use that procedure, enter the function name *sam_record* in this field.

Logs

KwikNet allows you to generate network information summaries called logs. These summaries are often referred to as network statistics. These logs can be dumped to your data logging device using KwikNet function *kn_netstats()*.

Check the box labeled "Network summary" if you want to be able to dump a brief summary of the current state of all network interfaces.

Check the box labeled "Device statistics" if you want to be able to dump the complete list of device statistics maintained by the device driver attached to each network interface.

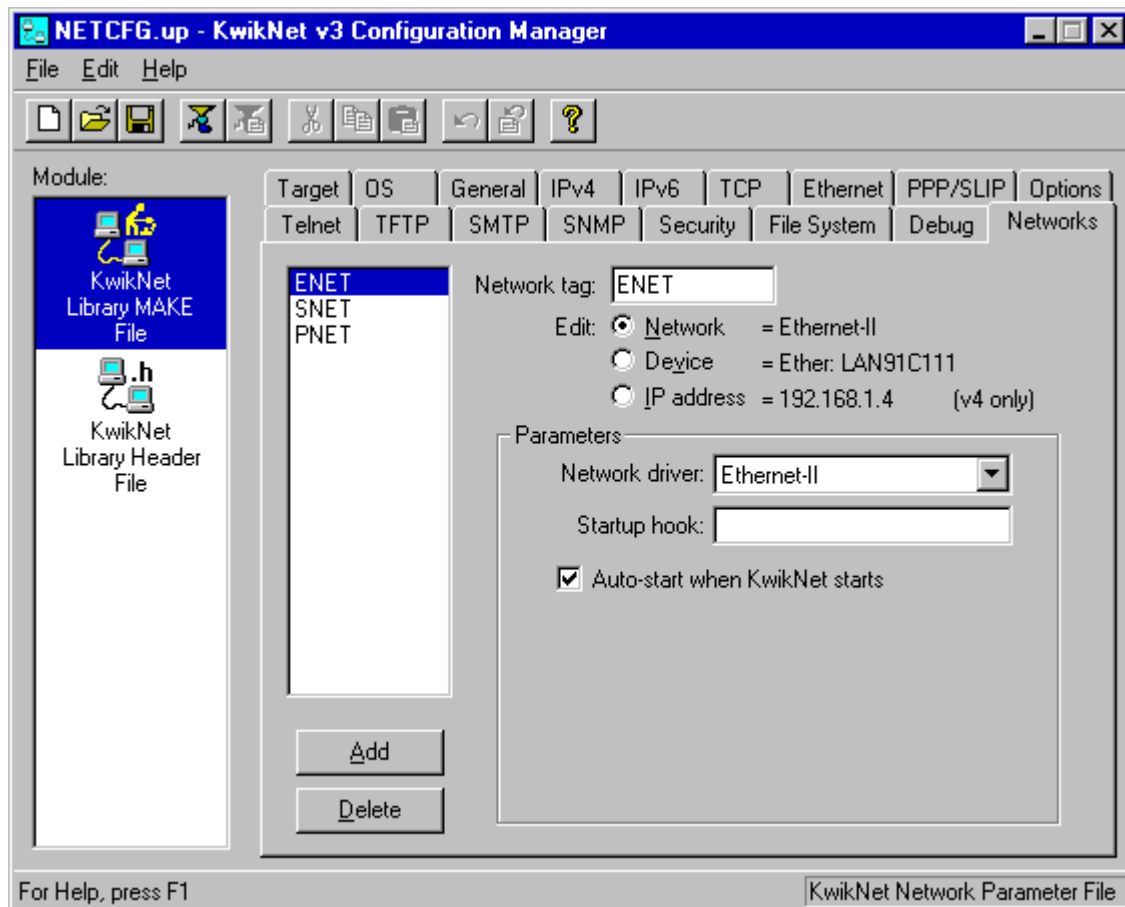
Check the box labeled "ARP, routing, TCP, UDP" if you want to be able to dump the complete list of network operating conditions available from the Turbo Trek TCP/IP Stack.

This page left blank intentionally.

2.4 Adding an Ethernet Network Interface

The easiest way to add an Ethernet network interface to your system is to let KwikNet do all of the work. You simply add a description of the network interface and its requirements to your KwikNet User Parameter File and KwikNet prebuilds it for you during its initialization sequence. Of course, you can always dynamically add Ethernet network interfaces at runtime after you start KwikNet. But for most applications, having the networks prebuilt and ready for use is the preferred solution.

To add a prebuilt Ethernet network interface definition to your KwikNet Library, use the KwikNet Configuration Manager to edit your Network Parameter File. A separate definition is required for each prebuilt network. The total number of prebuilt networks must not exceed the maximum number of networks which your KwikNet configuration allows. Each Ethernet network is defined using the Networks property page. The layout of the window is shown below.



Ethernet Network Definition (continued)

Tag

Each network must have a unique network tag. The tag is a string of 1 to 7 characters. This parameter defines that tag. Although KwikNet does not restrict the content of the tag in a network description, the Configuration Manager only supports 1 to 7 ASCII characters as a tag.

Edit: Network

You must select the Edit: Network radio button to define the network parameters.

Network Driver

You must select Ethernet-II or Ethernet 802.3 from the pull down list to attach the KwikNet Ethernet Network Driver to your Ethernet network.

You must add an Ethernet-II network if the interface uses Ethernet-II (DIX) framing exclusively. You must also enable the "Support Ethernet-II framing" option on the Ethernet property page. If this option is not enabled, your application's link will fail because function *kn_etnet_prep()* is not defined.

You must add an Ethernet 802.3 network if the interface uses 802.3 framing, possibly with Ethernet-II framing. You must also enable the "Support 802.3 framing" option on the Ethernet property page. If this option is not enabled, your application's link will fail because function *kn_et8023_prep()* is not defined. The framing mode used by all 802.3 interfaces is also specified on the Ethernet property page.

Startup Hook

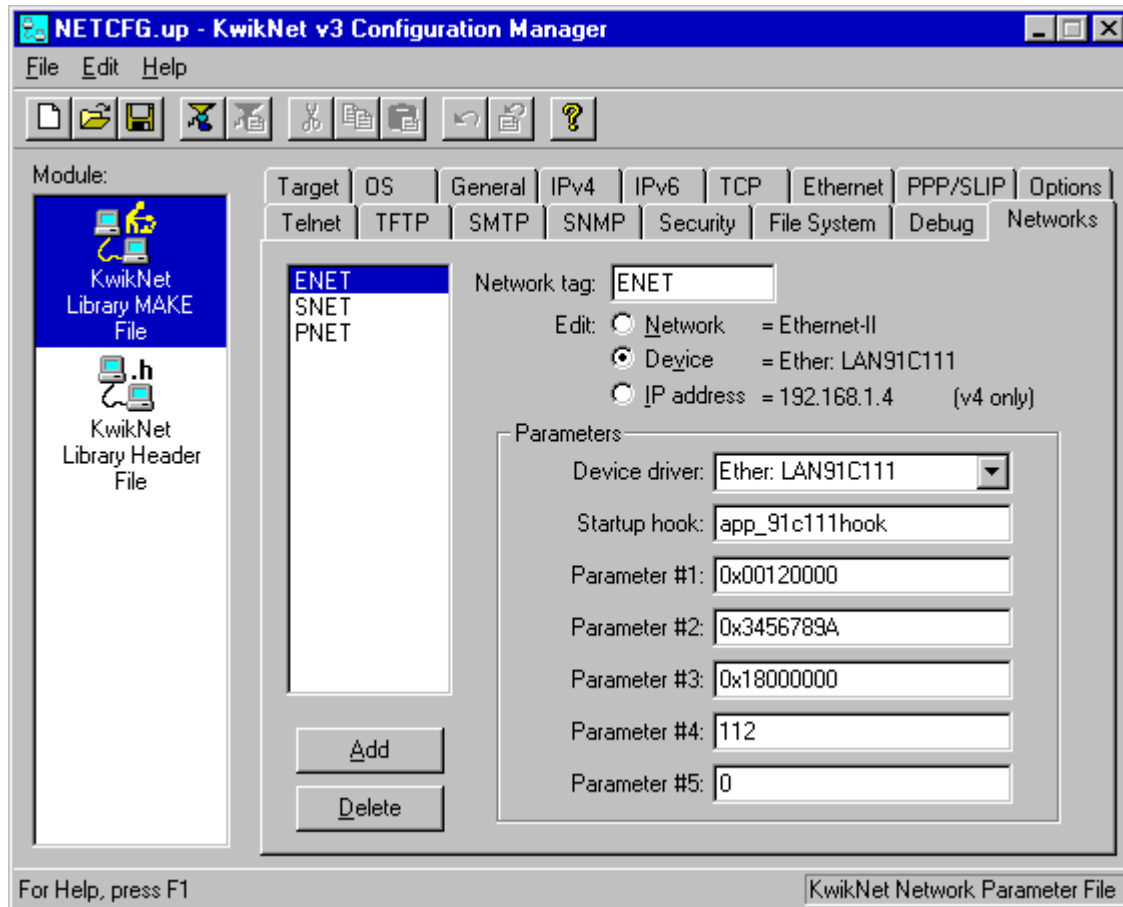
This parameter provides the name of an application function which will be called when the network driver is being initialized. This function can modify the network's configuration parameters and IP address information before the network interface has been fully initialized. If your application does not require a startup hook for this network, leave this field empty. The Ethernet network driver startup hook is described in Appendix A.1 of the KwikNet Device Driver Technical Reference Manual.

Auto Start

If this box is checked, the network interface will be automatically opened and made available for use as soon as KwikNet has started and prebuilt the network interface. If you leave this box unchecked, the network interface will be prebuilt by KwikNet but will be left closed until opened by your application with a call to KwikNet procedure *kn_ifopen()*.

Ethernet Device Driver Definition

You must define the device driver attached to each prebuilt network which your application supports. A separate device driver definition is required for each prebuilt network. The device driver for each prebuilt network is defined using the Networks property page. The layout of the window is shown below.



Ethernet Device Driver Definition (continued)

Tag

Each device driver inherits the unique network tag assigned to the network interface to which the device driver is attached. The "Network tag" field defines that tag.

Edit: Device

You must select the Edit: Device radio button to define the network's device driver parameters.

Device Driver

To use any of the KwikNet device drivers which are available from KADAK, select its name from the pull down list.

If you are using your own custom KwikNet device driver or one only recently available from KADAK, you must edit the text region of the list to identify the driver. Replace the text in the list box with the name of the device driver's Device Preparation Function *dddd_prep*. The string *dddd* in the function name is the mnemonic used to uniquely identify the particular device driver.

The selected device driver must match its network driver. Device drivers for Ethernet interface devices can only be used with the Ethernet Network Driver. Device drivers for serial (UART) interface devices can only be used with the SLIP or PPP Network Driver.

Startup Hook

This parameter provides the name of an application function which will be called when the device driver is being opened for use. This function can modify the device's configuration parameters before the device is actually made ready for use. If your application does not require a startup hook for this device, leave this field empty. The device driver startup hook is described in Appendix A.2 of the KwikNet Device Driver Technical Reference Manual.

Parameter #1 through #5

There are five optional parameters which can be used to configure the device driver. Each parameter can provide a 32-bit value. Unused parameters can be left empty. The use and meaning of each parameter is completely defined by the device driver.

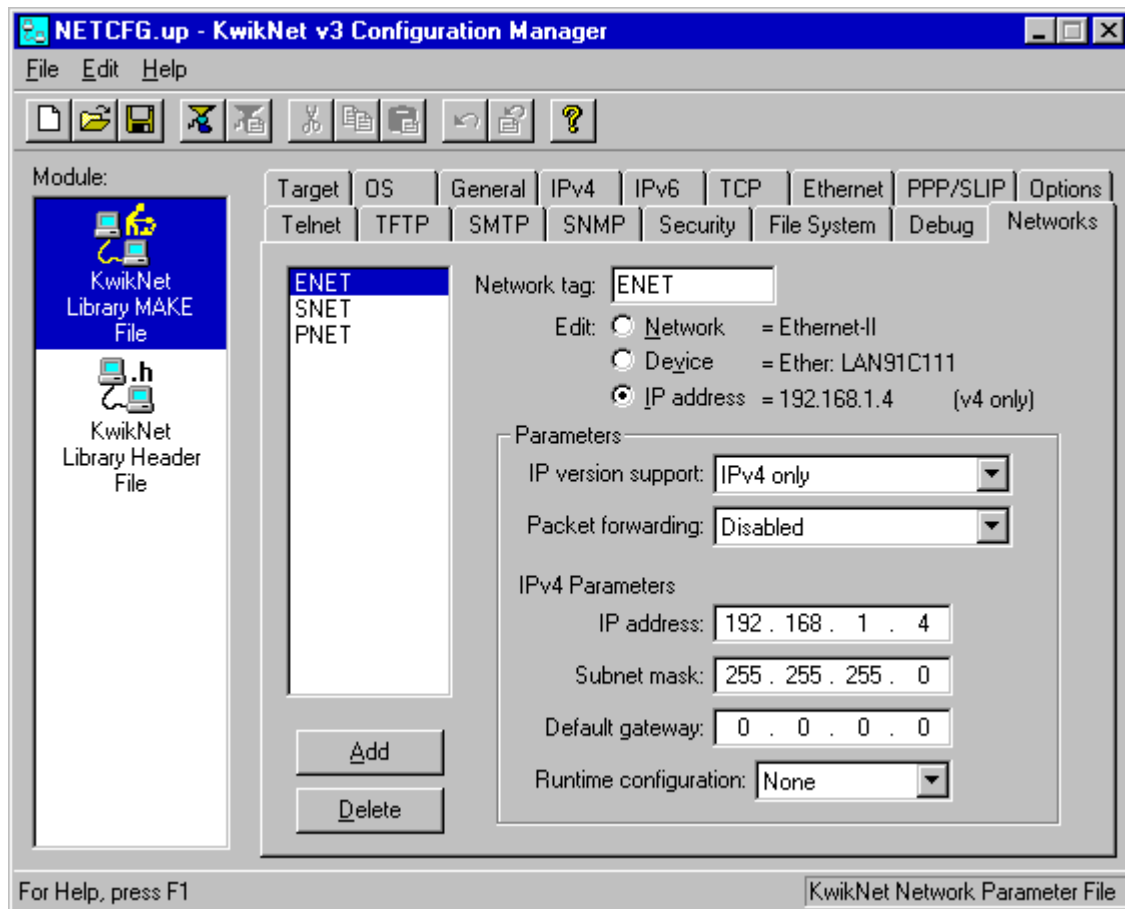
If you are using one of the KwikNet device drivers which are available from KADAK, refer to its manual for the driver's definition of these fields. Otherwise, refer to the data sheets which you created for your custom KwikNet device driver.

Ethernet IP Address Definition

You must provide a network IP address for each prebuilt network which your application supports. The network IP address for each prebuilt network is defined using the Networks property page. The layout of the window is shown below.

You can specify a network's IP address using this property page. Alternatively, you can assign the IP address in your network startup hook, if one is provided. Of course, IP addresses can also be assigned dynamically using DHCP, BOOTP or Auto IP.

If you intend to assign the network IP address at runtime, then you can enter 0.0.0.0 for all of the address fields on this page.



Edit: IP Address

You must select the Edit: IP Address radio button to define the network's IP address and related parameters.

Ethernet IP Address Definition (continued)

IP Version

Select the combination of IPv4 and IPv6 protocols that this network interface supports.

IPv4	Support only IPv4.
IPv6	Support only IPv6.
All (v4 and v6)	Both IPv4 and IPv6 must be supported.

If you select IPv6, be sure to configure your KwikNet Library to include IPv6 support. To do so, check the box labeled "Support IPv6" on the IPv6 property page. Failure to do so will result in a compilation error when you attempt to compile your KwikNet Library. The error will inform you that your library does not include support for IPv6.

Packet Forwarding

Select the type of packet forwarding that this network interface supports.

Disabled	Forwarding of packets is not allowed on this interface.
Unicast only	Support forwarding of IPv4 or IPv6 unicast packets.
Unicast + Dir. Broadcast	Support forwarding of IPv4 or IPv6 unicast packets and IPv4 directed broadcast packets.

IP Address

Enter the IP address of the network interface. This must be a unique, valid IP address which can be used to identify the host computer attached to this network interface. Enter 0.0.0.0 if the network IP address is to be assigned or dynamically detected at runtime.

Subnet Mask

Enter the subnet mask of the network. The subnet mask defines the manner in which IP addresses on this network are decoded to distinguish between the physical net address and specific host identifiers residing at that physical net address. Enter 0.0.0.0 if subnet addressing is not used on this network or is to be assigned or dynamically detected at runtime.

Default Gateway

Enter the IP address of the default gateway to be used in the absence of specific routing information. This value must be a valid host IP address or 0.0.0.0. Enter 0.0.0.0 if the network has no default gateway or if the default gateway is to be assigned or dynamically detected at runtime.

The default gateway is used to identify the network interface through which packets are to be sent if the destination IP address is not otherwise known. Such packets will be sent out this interface using the host on this network identified as the default gateway.

There can only be one default gateway in the system. The first network interface to successfully declare its gateway address as the default will be used as the gateway.

Ethernet IP Address Definition (continued)

Runtime Configuration

This parameter is used to select one of the several possible methods for dynamically deriving the network IP address, subnet mask and default gateway for your network. Choose the method from the pull down list.

Select **None** if these parameters are to be established according to the definitions which you have entered into the fields on this property page.

Select **DHCP** if the network interface is to use the Dynamic Host Configuration Protocol to derive an IP address, subnet mask and gateway definition for the network. To use this option, there must be a DHCP server located on this network.

The **DHCP (reboot)** option is identical to the DHCP option except that the DHCP initialization sequence always starts in the *REBOOT* state when the interface is opened, allowing the DHCP server to assign the same IP address to your network after it has been down briefly.

If you select DHCP or DHCP (reboot), be sure to configure your KwikNet Library to include the KwikNet DHCP Client. To do so, check the box labeled "DHCP Client support" on the IPv4 property page. Failure to do so will result in a compilation error when you attempt to compile your KwikNet Library. The error will inform you that your library does not include the DHCP Client.

Select **BOOTP** if the network interface is to use the older BOOT Protocol to derive an IP address, subnet mask and gateway definition for the network. To use this option, there must be a BOOTP server located on this network.

Select **Auto IP** if the network interface is to use the optional KwikNet Auto IP component to derive an IP address for the network. The subnet mask will be set to 255.255.0.0 and the default gateway will be set according to the definitions which you have entered on this property page.

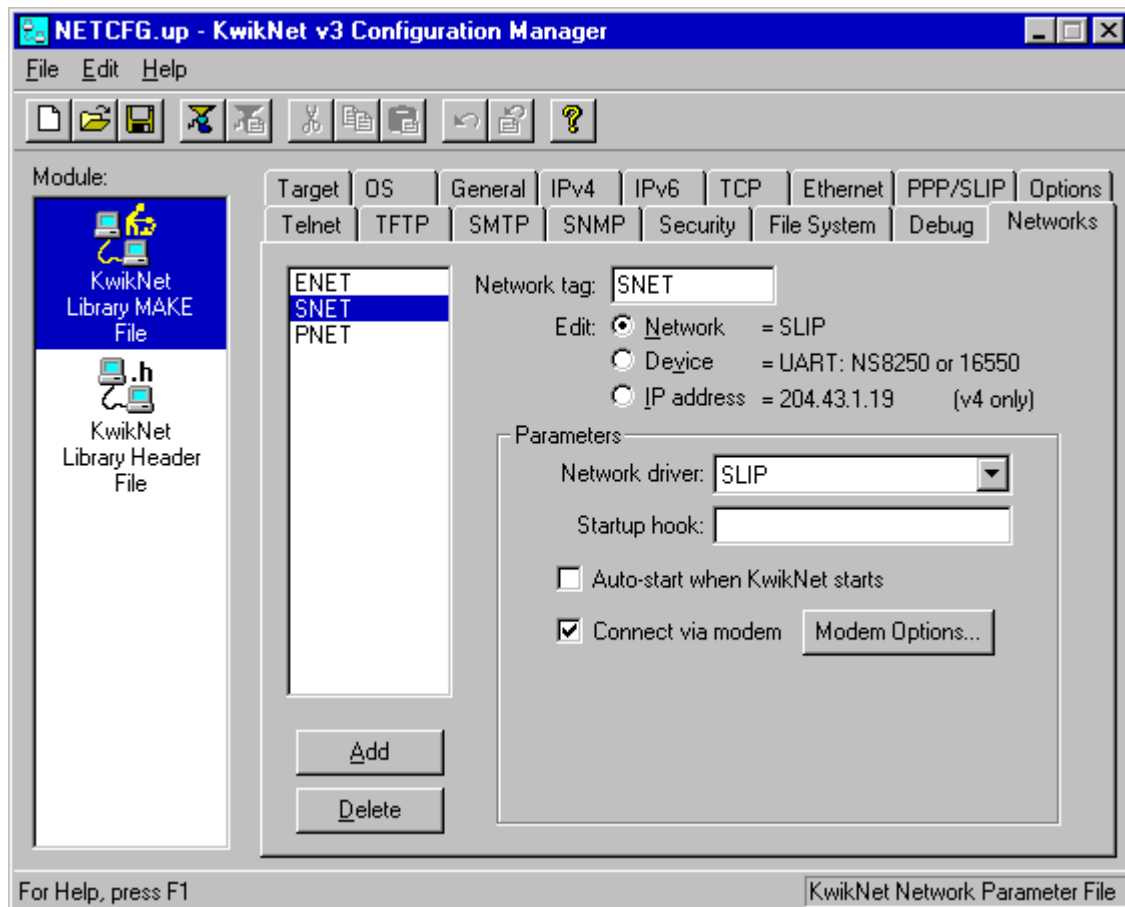
If you select Auto IP, be sure to configure your KwikNet Library to include Auto IP support. To do so, check the box labeled "Auto IP address discovery" on the IPv4 property page. Failure to do so will result in a compilation error when you attempt to compile your KwikNet Library. The error will inform you that your library does not include support for Auto IP.

This page left blank intentionally.

2.5 Adding a SLIP Network Interface

The easiest way to add a SLIP network interface to your system is to let KwikNet do all of the work. You simply add a description of the network interface and its requirements to your KwikNet User Parameter File and KwikNet prebuilds it for you during its initialization sequence. Of course, you can always dynamically add SLIP network interfaces at runtime after you start KwikNet. But for most applications, having the networks prebuilt and ready for use is the preferred solution.

To add a prebuilt SLIP network interface definition to your KwikNet Library, use the KwikNet Configuration Manager to edit your Network Parameter File. A separate definition is required for each prebuilt network. The total number of prebuilt networks must not exceed the maximum number of networks which your KwikNet configuration allows. Each SLIP network is defined using the Networks property page. The layout of the window is shown below.



SLIP Network Definition (continued)

Tag

Each network must have a unique network tag. The tag is a string of 1 to 7 characters. This parameter defines that tag. Although KwikNet does not restrict the content of the tag in a network description, the Configuration Manager only supports 1 to 7 ASCII characters as a tag.

Edit: Network

You must select the Edit: Network radio button to define the network parameters.

Network Driver

You must select SLIP from the pull down list to attach the KwikNet SLIP Network Driver to your SLIP network.

Startup Hook

This parameter provides the name of an application function which will be called when the network driver is being initialized. This function can modify the network's configuration parameters and IP address information before the network interface has been fully initialized. If your application does not require a startup hook for this network, leave this field empty. The SLIP network driver startup hook is described in Appendix A.1 of the KwikNet Device Driver Technical Reference Manual.

Auto Start

If this box is checked, the network interface will be automatically opened and made available for use as soon as KwikNet has started and prebuilt the network interface. If you leave this box unchecked, the network interface will be prebuilt by KwikNet but will be left closed until opened by your application with a call to KwikNet procedure `kn_ifopen()`.

Modem Connection

The SLIP network driver supports remote connections using the KwikNet Modem Driver. Check this box to attach the modem driver to this network. Otherwise, leave this box unchecked. The KwikNet Modem Driver is described in Chapter 1.7 of the KwikNet Device Driver Technical Reference Manual.

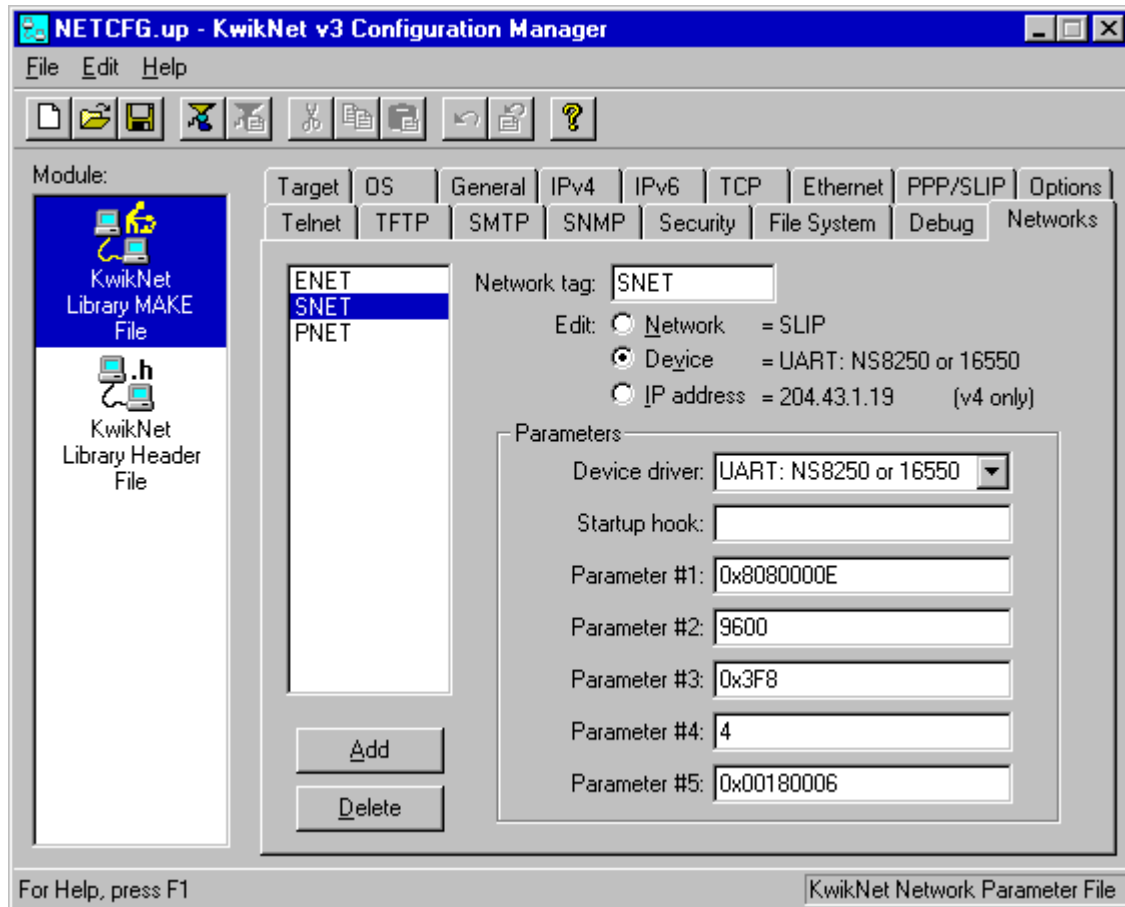
Note that the network still requires a device driver even if the modem driver is used.

Modem Options

If you have attached the modem driver to this network, then click the Modem Options... button to open the Modem Options Dialog. Within this dialog you can configure the modem to meet your requirements (see Chapter 2.7).

SLIP Serial Device Driver Definition

You must define the device driver attached to each prebuilt network which your application supports. A separate device driver definition is required for each prebuilt network. The device driver for each prebuilt network is defined using the Networks property page. The layout of the window is shown below.



SLIP Serial Device Driver Definition (continued)

Tag

Each device driver inherits the unique network tag assigned to the network interface to which the device driver is attached. The "Network tag" field defines that tag.

Edit: Device

You must select the Edit: Device radio button to define the network's device driver parameters.

Device Driver

To use any of the KwikNet device drivers which are available from KADAK, select its name from the pull down list.

If you are using your own custom KwikNet device driver or one only recently available from KADAK, you must edit the text region of the list to identify the driver. Replace the text in the list box with the name of the device driver's Device Preparation Function *dddd_prep*. The string *dddd* in the function name is the mnemonic used to uniquely identify the particular device driver.

The selected device driver must match its network driver. Device drivers for serial (UART) interface devices can only be used with the SLIP or PPP Network Driver. Device drivers for Ethernet interface devices can only be used with the Ethernet Network Driver.

Startup Hook

This parameter provides the name of an application function which will be called when the device driver is being opened for use. This function can modify the device's configuration parameters before the device is actually made ready for use. If your application does not require a startup hook for this device, leave this field empty. The device driver startup hook is described in Appendix A.2 of the KwikNet Device Driver Technical Reference Manual.

Parameter #1 through #5

There are five optional parameters which can be used to configure the device driver. Each parameter can provide a 32-bit value. Unused parameters can be left empty. The use and meaning of each parameter is completely defined by the device driver.

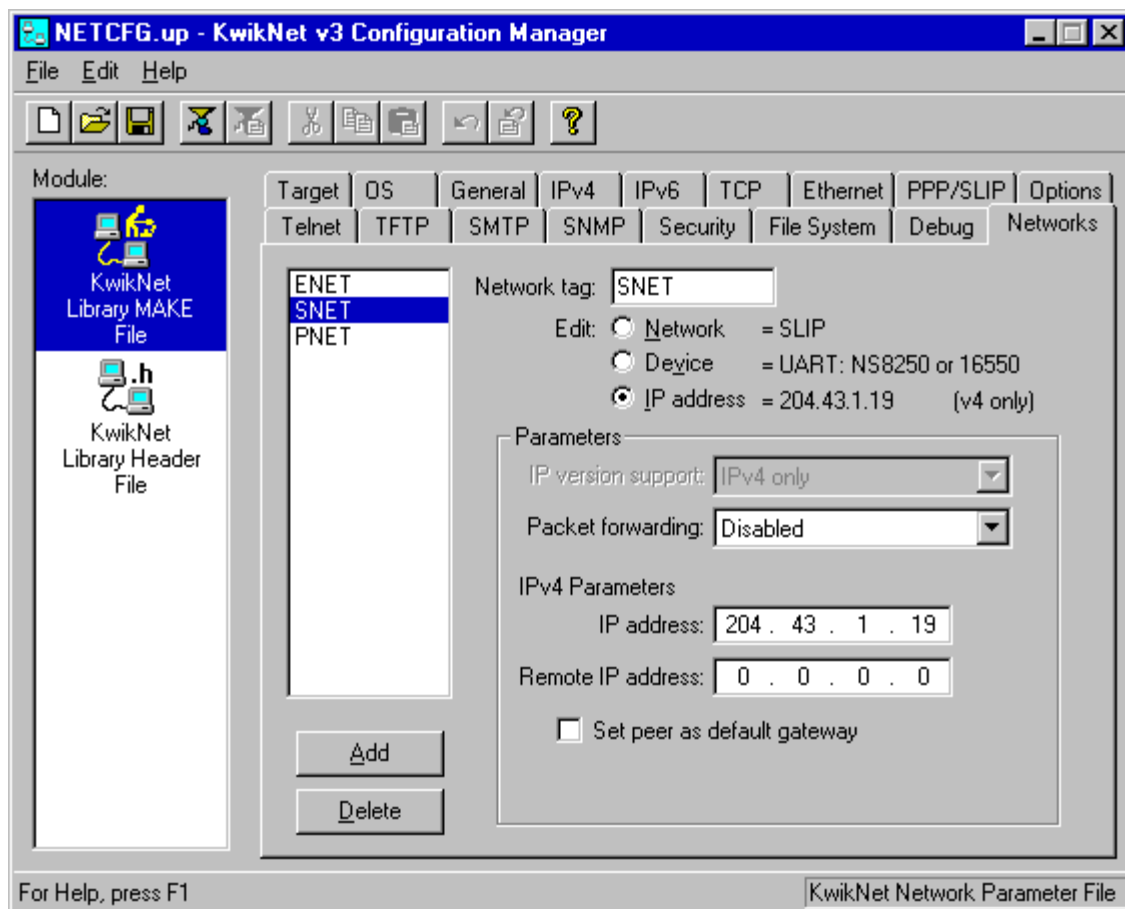
If you are using one of the KwikNet device drivers which are available from KADAK, refer to its manual for the driver's definition of these fields. Otherwise, refer to the data sheets which you created for your custom KwikNet device driver.

SLIP IP Address Definition

You must provide a network IP address for each prebuilt network which your application supports. The network IP address for each prebuilt network is defined using the Networks property page. The layout of the window is shown below.

You can specify a network's IP address using this property page. Alternatively, you can assign the IP address in your network startup hook, if one is provided. KwikNet does not support the dynamic assignment of an IP address for a SLIP network using DHCP, BOOTP or Auto IP.

If you intend to assign the network IP address at runtime, then you can enter 0.0.0.0 for all of the address fields on this page.



SLIP IP Address Definition (continued)

Edit: IP Address

You must select the Edit: IP Address radio button to define the network's IP address and related parameters.

IP Version

SLIP only supports IPv4. Therefore, the IP version selection is not used.

Packet Forwarding

Select the type of packet forwarding that this network interface supports.

Disabled	Forwarding of packets is not allowed on this interface.
Unicast only	Support forwarding of IPv4 unicast packets.
Unicast + Dir. Broadcast	Support forwarding of IPv4 unicast packets and IPv4 directed broadcast packets.

IP Address

Enter the IP address of the network interface. This must be a unique, valid IP address which can be used to identify the host computer attached to this network interface. Enter 0.0.0.0 if the network IP address is to be assigned at runtime.

Remote IP Address

Enter the IP address of the foreign host to which the SLIP network interface is to be connected. Enter 0.0.0.0 if the remote IP address is to be assigned at runtime.

Default Gateway

Check this box if you want the IP address of the foreign host to which the SLIP network interface is connected to be used as the default gateway. The default gateway will be set after the SLIP network interface has been opened and successfully connected to its peer.

The default gateway is used to identify the network interface through which packets are to be sent if the destination IP address is not otherwise known. Such packets will be sent out this SLIP network interface using the remote peer as the default gateway.

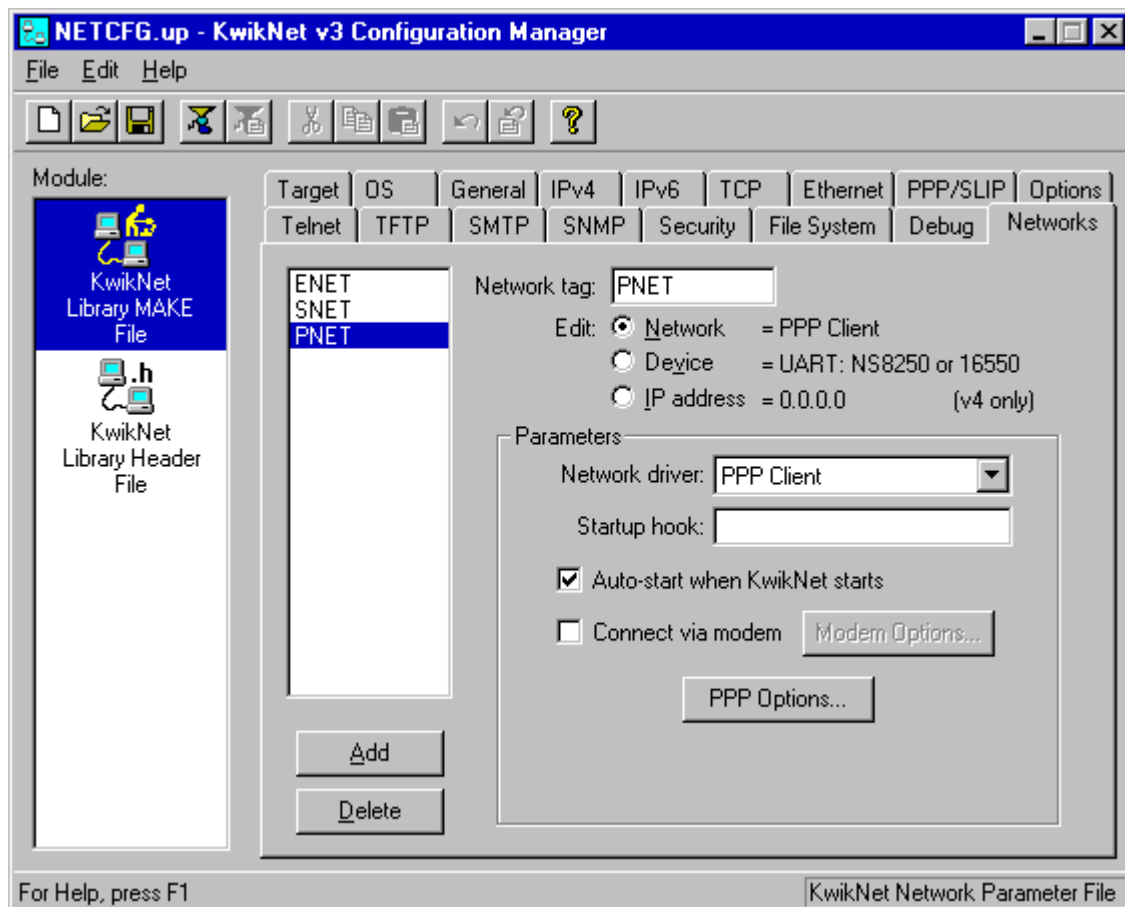
There can only be one default gateway in the system. The first network interface to successfully declare its gateway address as the default will be used as the gateway.

2.6 Adding a PPP Network Interface

The easiest way to add a PPP network interface to your system is to let KwikNet do all of the work. You simply add a description of the network interface and its requirements to your KwikNet User Parameter File and KwikNet prebuilds it for you during its initialization sequence. Of course, you can always dynamically add PPP network interfaces at runtime after you start KwikNet. But for most applications, having the networks prebuilt and ready for use is the preferred solution.

There are two kinds of PPP networks: client and server. You must use a PPP Client network if you need to initiate the connection to your PPP peer. You must use a PPP Server network if you intend to accept requests for connection from your PPP peer. If you must operate in either mode, you must provide two networks, one for your PPP client and one for your PPP server. You must then ensure that one or the other, but not both, is in control of the serial network interface at any particular instant.

To add a prebuilt PPP network interface definition to your KwikNet Library, use the KwikNet Configuration Manager to edit your Network Parameter File. A separate definition is required for each prebuilt network. The total number of prebuilt networks must not exceed the maximum number of networks which your KwikNet configuration allows. Each PPP network is defined using the Networks property page. The layout of the window is shown below.



PPP Network Definition (continued)

Tag

Each network must have a unique network tag. The tag is a string of 1 to 7 characters. This parameter defines that tag. Although KwikNet does not restrict the content of the tag in a network description, the Configuration Manager only supports 1 to 7 ASCII characters as a tag.

Edit: Network

You must select the Edit: Network radio button to define the network parameters.

Network Driver

You must select PPP Client or PPP Server from the pull down list to attach the KwikNet PPP Network Driver to your PPP network. You must add a PPP Client network if you need to initiate the connection to your PPP peer. You must add a PPP Server network if you intend to accept requests for connection from your PPP peer.

Note: If you must operate in either mode, you must provide two networks, one for your PPP client and one for your PPP server. You must then ensure that one or the other, but not both, is in control of the serial network interface at any particular instant.

Startup Hook

This parameter provides the name of an application function which will be called when the network driver is being initialized. This function can modify the network's configuration parameters and IP address information before the network interface has been fully initialized. If your application does not require a startup hook for this network, leave this field empty. The PPP network driver startup hook is described in Appendix A.1 of the KwikNet Device Driver Technical Reference Manual.

Auto Start

If this box is checked, the network interface will be automatically opened and made available for use as soon as KwikNet has started and prebuilt the network interface. If you leave this box unchecked, the network interface will be prebuilt by KwikNet but will be left closed until opened by your application with a call to KwikNet procedure `kn_ifopen()`.

Note: If you prebuild a PPP client and a PPP server network interface which use a common serial link, you can auto-start one or the other, but not both, as described in Chapter 6.3.

PPP Network Definition (continued)

Modem Connection

The PPP network driver supports remote connections using the KwikNet Modem Driver. Check this box to attach the modem driver to this network. Otherwise, leave this box unchecked. The KwikNet Modem Driver is described in Chapter 1.7 of the KwikNet Device Driver Technical Reference Manual.

Note that the network still requires a device driver even if the modem driver is used.

Modem Options

If you have attached the modem driver to this network, then click the Modem Options... button to open the Modem Options Dialog. Within this dialog you can configure the modem to meet your requirements (see Chapter 2.7).

PPP Network Definition (continued)

PPP Options

Click on the PPP Options... button to open the PPP Options Dialog. Within this dialog you can adjust LCP, authentication and IPCP parameters to meet your PPP requirements. Note that these PPP option settings are for one particular PPP network interface. Different PPP networks can have different option settings.

The layout of the PPP dialog box is shown below.

The screenshot shows the 'PPP Network Options' dialog box with a blue title bar and a close button. It is divided into three main sections: LCP Negotiation, Authentication, and IPCP Negotiation.

LCP Negotiation

- ☒ Header field compression
- ☒ Magic number negotiation (turn off for loopback)

Authentication

Require		Accept
<input type="checkbox"/> <---	EAP	---> <input type="checkbox"/>
<input type="checkbox"/> <---	MS-CHAP v1	---> <input type="checkbox"/>
<input type="checkbox"/> <---	CHAP	---> <input checked="" type="checkbox"/>
<input type="checkbox"/> <---	PAP	---> <input checked="" type="checkbox"/>

Require = Peer must authenticate itself using one of these protocols.
Accept = Local network will authenticate using one of these protocols.

IPCP Negotiation

- ☒ IP compression
- ☒ Request DNS server information from peer
- ☒ Set peer as default gateway

Local IP address: 0 . 0 . 0 . 0

Remote IP address: 0 . 0 . 0 . 0

Primary DNS server: 0 . 0 . 0 . 0

Secondary DNS server: 0 . 0 . 0 . 0

At the bottom are 'OK' and 'Cancel' buttons.

PPP Network Definition (continued)

LCP Negotiation Options

Header Compression

Check this box if the PPP network is permitted to negotiate the use of header compression to reduce the length of the PPP framing header. For most PPP networks, this box should be checked. If you know that the network's peer does *not* support header compression, leave this box unchecked to avoid the unnecessary negotiation.

Magic Number Negotiation

Check this box if magic numbers are to be used during LCP negotiation. Otherwise, leave this box unchecked. Be sure to leave this box unchecked if you are using a loopback device driver.

Authentication Options

Require/Accept

KwikNet supports the four authentication protocols listed on this property page. If this network interface demands that its peer authenticate itself using any of these protocols, check the box in the column labeled Require. If this network interface will accept a request from its peer to authenticate this interface using any of these protocols, check the box in the column labeled Accept.

For each protocol, either Require or Accept or both or neither can be checked. If more than one protocol is listed as required, the network interface will attempt to authenticate its peer using each checked protocol in the order listed (EAP to PAP) until successful with one of the protocols or until all have failed. If more than one protocol is listed as accepted, the network interface will authenticate itself if, and only if, its peer requests authentication using one of the checked protocols.

Check the **EAP** boxes only if this network interface supports the Extensible Authentication Protocol (EAP).

Check the **MS-CHAP v1** boxes only if this network interface supports the Microsoft Challenge-Handshake Authentication Protocol (MS-CHAP).

Check the **CHAP** boxes only if this network interface supports the standard Challenge-Handshake Authentication Protocol (CHAP).

Check the **PAP** boxes only if this network interface supports the Password Authentication Protocol (PAP).

Note: If EAP or MS-CHAP support has not been enabled on the PPP/SLIP property page, then the corresponding PPP authentication options will be ignored.

PPP Network Definition (continued)

IPCP Negotiation Options

IP Compression

Check this box if the PPP network is permitted to negotiate the use of IP header compression. Either Van Jacobson or IPHC header compression will be used, according to your selection on the PPP/SLIP property page (see Chapter 2.3). If the PPP network must not use IP header compression, leave this box unchecked.

Request DNS Server

Check this box if you will accept a primary or secondary DNS server IP address for use by your DNS Client if offered by the network's peer. Otherwise, leave this box unchecked. If you check this box, be sure to include the KwikNet DNS Client by checking the box labeled "Use DNS Client" on the Options property page (see Chapter 2.3).

Default Gateway

The default gateway for each PPP network is usually established by checking the option in the network's IP address definition which you access by selecting the Edit: IP Address radio button on the Networks property page. For convenience, the default gateway option can also be set using this PPP Options Dialog.

Check this box if you want the IP address of the foreign host to which the PPP network interface is connected to be used as the default gateway. The default gateway will be set after the PPP network interface has been opened and successfully connected to its peer. Otherwise, leave this box unchecked.

PPP Options (continued)

Local IP Address

The IP address for each PPP network is usually edited in the network's IP address definition which you access by selecting the Edit: IP Address radio button on the Networks property page. For convenience, the network IP address can also be edited using this PPP Options Dialog.

PPP allows the local IP address to be negotiated with the network's peer.

If the local IP address is 0.0.0.0, then the local IP address must be negotiated with the peer. In this case, the peer must provide the local IP address if the network is to be operable.

If a local IP address is provided, then no negotiation of the local IP address will be permitted. The IP address which you defined will be used as the local IP address.

Remote IP Address

The IP address of the PPP network's peer is usually edited in the network's IP address definition which you access by selecting the Edit: IP Address radio button on the Networks property page. For convenience, the remote IP address can also be edited using this PPP Options Dialog.

PPP allows the IP address of the network's peer to be negotiated with the peer, if permitted by the peer. If no such negotiation is required, set the remote IP address to 0.0.0.0 in the space provided.

If a remote IP address is provided, it will be presented for acceptance by the peer if, and only if, the peer requests that it be provided with an IP address. If the peer rejects the proposal, the IP address of the peer will be as defined by the peer. In this case, if the peer does not have an alternative IP address, the network will be inoperable.

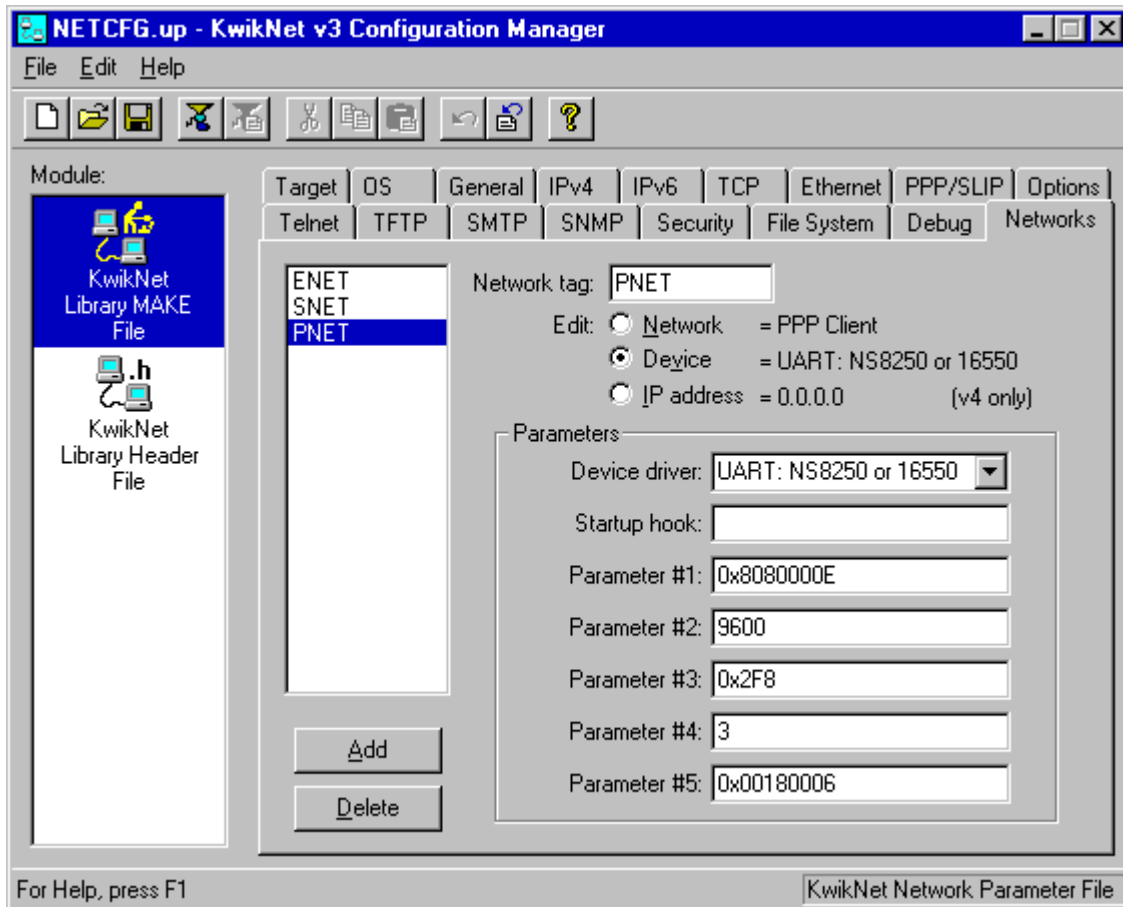
If the peer asks to negotiate its IP address and the defined remote IP address is 0.0.0.0, then the defined local IP address plus one will be proposed for use by the peer, provided that the defined local IP address is not 0.0.0.0.

DNS Server IP Addresses

KwikNet can send the IP address of one or two DNS servers to the network's peer for use by that peer. You can enter the primary and/or secondary DNS server IP address in the space provided. An IP address of 0.0.0.0 will not be presented to the peer during DNS negotiation.

PPP Serial Device Driver Definition

You must define the device driver attached to each prebuilt network which your application supports. A separate device driver definition is required for each prebuilt network. The device driver for each prebuilt network is defined using the Networks property page. The layout of the window is shown below.



PPP Serial Device Driver Definition (continued)

Tag

Each device driver inherits the unique network tag assigned to the network interface to which the device driver is attached. The "Network tag" field defines that tag.

Edit: Device

You must select the Edit: Device radio button to define the network's device driver parameters.

Device Driver

To use any of the KwikNet device drivers which are available from KADAK, select its name from the pull down list.

If you are using your own custom KwikNet device driver or one only recently available from KADAK, you must edit the text region of the list to identify the driver. Replace the text in the list box with the name of the device driver's Device Preparation Function *dddd_prep*. The string *dddd* in the function name is the mnemonic used to uniquely identify the particular device driver.

The selected device driver must match its network driver. Device drivers for serial (UART) interface devices can only be used with the SLIP or PPP Network Driver. Device drivers for Ethernet interface devices can only be used with the Ethernet Network Driver.

Startup Hook

This parameter provides the name of an application function which will be called when the device driver is being opened for use. This function can modify the device's configuration parameters before the device is actually made ready for use. If your application does not require a startup hook for this device, leave this field empty. The device driver startup hook is described in Appendix A.2 of the KwikNet Device Driver Technical Reference Manual.

Parameter #1 through #5

There are five optional parameters which can be used to configure the device driver. Each parameter can provide a 32-bit value. Unused parameters can be left empty. The use and meaning of each parameter is completely defined by the device driver.

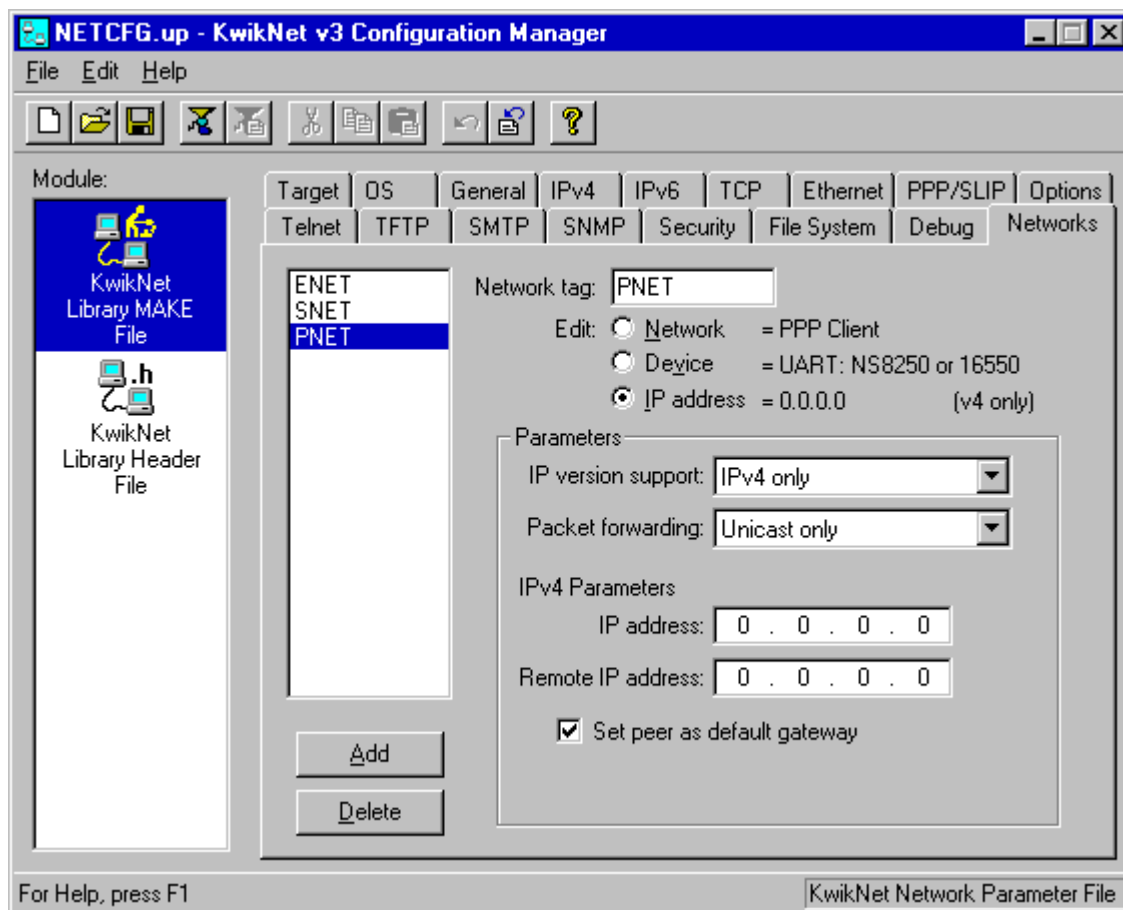
If you are using one of the KwikNet device drivers which are available from KADAK, refer to its manual for its definition of these fields. Otherwise, refer to the data sheets which you created for your custom KwikNet device driver.

PPP IP Address Definition

You must provide a network IP address for each prebuilt network which your application supports. The network IP address for each prebuilt network is defined using the Networks property page. The layout of the window is shown below.

You can specify a network's IP address using this property page. Alternatively, you can assign the IP address in your network startup hook, if one is provided, or negotiate the IP address with the network's PPP peer. KwikNet does not support the dynamic assignment of an IP address for a PPP network using DHCP, BOOTP or Auto IP.

If you intend to assign or negotiate the network IP address at runtime, then you can enter 0.0.0.0 for all of the address fields on this page. For your convenience, all the options on this page can also be edited on the more comprehensive PPP Options Dialog which you access by selecting the Edit: Network radio button and then clicking on the PPP Options... button.



PPP IP Address Definition (continued)

Edit: IP Address

You must select the Edit: IP Address radio button to define the network's IP address and related parameters.

IP Version

Select the combination of IPv4 and IPv6 protocols that this network interface supports.

IPv4	Support only IPv4.
IPv6	Support only IPv6.
All (v4 and v6)	Both IPv4 and IPv6 must be supported.

If you select IPv6, be sure to configure your KwikNet Library to include IPv6 support. To do so, check the box labeled "Support IPv6" on the IPv6 property page. Failure to do so will result in a compilation error when you attempt to compile your KwikNet Library. The error will inform you that your library does not include support for IPv6.

Packet Forwarding

Select the type of packet forwarding that this network interface supports.

Disabled	Forwarding of packets is not allowed on this interface.
Unicast only	Support forwarding of IPv4 or IPv6 unicast packets.
Unicast + Dir. Broadcast	Support forwarding of IPv4 or IPv6 unicast packets and IPv4 directed broadcast packets.

IP Address

Enter the IP address of the network interface. This must be a unique, valid IP address which can be used to identify the host computer attached to this network interface. Enter 0.0.0.0 if the network IP address is to be assigned or negotiated at runtime.

Remote IP Address

Enter the proposed IP address for the foreign host to which the PPP network interface is to be connected. If an address is provided, the network driver will attempt to negotiate with the peer to establish the specified address as the peer's IP address. If the negotiation fails, the network will accept the address provided by the peer. Enter 0.0.0.0 if the proposed remote IP address is to be assigned at runtime or if there is no desire to propose an IP address for use by the peer.

PPP IP Address Definition (continued)

Default Gateway

Check this box if you want the IP address of the foreign host to which the PPP network interface is connected to be used as the default gateway. The default gateway will be set after the PPP network interface has been opened and successfully connected to its peer.

The default gateway is used to identify the network interface through which packets are to be sent if the destination IP address is not otherwise known. Such packets will be sent out this PPP network interface using the remote peer as the default gateway.

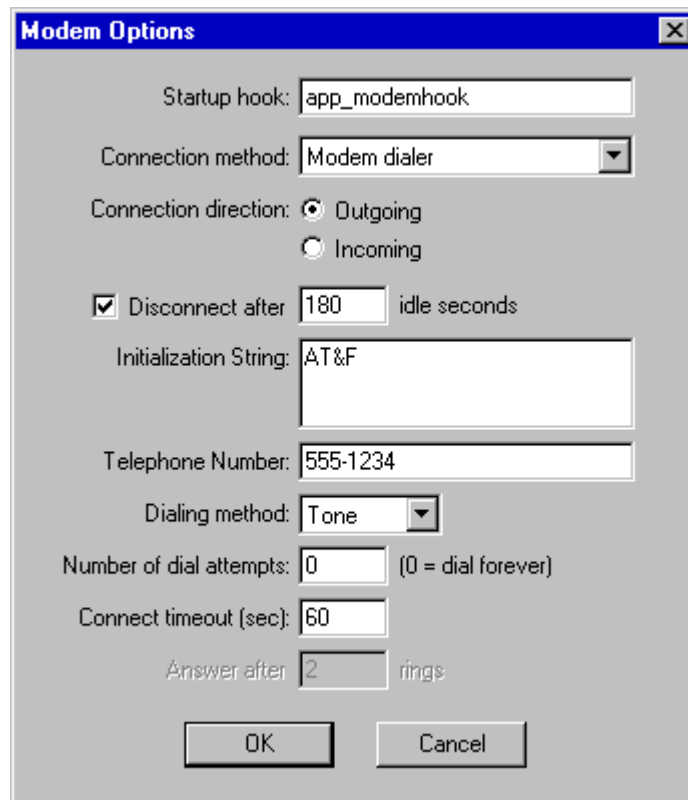
There can only be one default gateway in the system. The first network interface to successfully declare its gateway address as the default will be used as the gateway.

Note

The IP address, remote IP address and default gateway parameters for this PPP network can also be edited using the PPP Options Dialog. Select the Edit: Network radio button and click on the PPP Options... button.

2.7 Modem Options

If a KwikNet network requires modem support, you must define the modem parameters which govern its use. These modem parameters are edited using the Modem Options Dialog which you access by selecting the Edit: Network radio button and then clicking on the Modem Options... button. The layout of the dialog box is shown below.



The image shows a Windows-style dialog box titled "Modem Options". It contains several configuration fields: "Startup hook" with the text "app_modemhook"; "Connection method" with a dropdown menu showing "Modem dialer"; "Connection direction" with two radio buttons, "Outgoing" (selected) and "Incoming"; a checked checkbox "Disconnect after" followed by a text box "180" and the text "idle seconds"; "Initialization String" with a text box containing "AT&F"; "Telephone Number" with a text box containing "555-1234"; "Dialing method" with a dropdown menu showing "Tone"; "Number of dial attempts" with a text box "0" and the text "(0 = dial forever)"; "Connect timeout (sec)" with a text box "60"; and "Answer after" with a text box "2" and the text "rings". At the bottom are "OK" and "Cancel" buttons.

Startup Hook

This parameter provides the name of an application function which will be called when the KwikNet Modem Driver is being initialized. This function can modify the modem's configuration parameters before the modem has been fully initialized. If your application does not require a startup hook for this modem, leave this field empty. The modem driver startup hook is described in Appendix A.3 of the KwikNet Device Driver Technical Reference Manual.

Modem Options (continued)

Connection Method

From the pull down list, choose the method by which the modem connection is to be established.

Select **Modem dialer** if the connection will be established using a modem which is initialized using conventional modem commands and their expected modem responses. An optional login script can be used to complete the connection with the remote system after the modem connection has been established. Login scripts are described in Chapter 1.7 of the KwikNet Device Driver Technical Reference Manual.

Select **Login script** if the connection will be established using a login script which defines the sequence of commands to be issued and the response expected. You must provide a login script to use this method. All modem initialization and remote login must be handled by your login script. Login scripts are described in Chapter 1.7 of the KwikNet Device Driver Technical Reference Manual.

Select **Windows direct connection** if you are using a direct wired connection to a host computer running the Microsoft Windows operating system. This direct connection is considered a pseudo modem because Windows requires a special login string to establish the connection. This method of connection does not support login scripts.

Connection Direction

Once you have selected the connection method, select the radio button to indicate the direction in which the connection will be established.

Select the **Outgoing** radio button if your modem must initiate the connection.

Select the **Incoming** radio button if the remote system must initiate the connection.

Disconnect When Idle

If you want to automatically disconnect from the remote system when the connection has not been used recently, check this box and enter the number of seconds that the line must remain idle before disconnecting. If you leave this box unchecked, the line will remain active until the remote system terminates the connection, your application calls `kn_ifclose()` to shut down the network interface to which the modem is attached or KwikNet is shut down.

Note

Once you have selected the connection method and direction, the relevant dialing parameters will be editable. All other parameters will appear dimmed.

Modem Options (continued)

Initialization String

The modem initialization string is sent to the modem before each dialing attempt. The command resets the modem and prepares it for dialing. This string should include the Hayes standard "AT" command prefix. Some common initialization strings are "AT&F" and "ATZ". For more information, consult your modem's reference manual.

Telephone Number

Enter the telephone number of the remote system to which the modem driver will connect. This string may contain special characters that are used to modify the modem's dialing operation. Do not include the Hayes standard "ATDx" prefix in this string.

Some common dialing commands are listed below. Be sure to consult your modem's reference manual for the actual commands that it supports.

<i>0</i> through <i>9</i>	Dial this number
<i>() - space</i>	Brackets, minus and space characters are ignored
<i>comma</i>	Timed wait
<i>W</i>	Timed wait for a dial tone
<i>@</i>	Timed wait for remote pickup
<i>S</i>	Dial a number stored in the modem's internal memory

Dialing Method

From the pull down list, choose the dialing method to use with the attached telephone system. Most telephone systems support **Tone** dialing but some older systems may only support **Pulse** dialing.

Number of Dial Attempts

Enter the number of times that the modem driver will attempt to connect to the remote system before declaring a connection error. Note that a Windows direct connection is not actually dialed but the modem driver will make multiple attempts to connect. If you specify *0* attempts, the modem driver will retry after every connection timeout.

Connection Timeout

This field defines the maximum number of seconds that the modem driver will wait for a modem response after it has started a connection attempt. If the modem does not respond within this time, the dialing attempt will be aborted.

Auto-answer

If the modem driver is configured to answer incoming calls (direction is Incoming), enter the number of rings to wait before answering the call. The value must be positive and non-zero.

This page left blank intentionally.

3. KwikNet System Construction

3.1 Building an Application

If you are using KwikNet with AMX or have ported KwikNet to your operating environment, you are now ready to construct the KwikNet Library and build an actual KwikNet application. The sample program(s) provided with KwikNet and its optional components are working examples which you can use either for guidance or as a starting point for your own application.

To build a KwikNet application you must perform the following steps.

1. If none of the available KwikNet device drivers meet your needs, create a custom device driver as described in the KwikNet Device Driver Technical Reference Manual.
2. If necessary, adapt the KwikNet board driver *KN_BOARD.C* to accommodate your target processor, device interfaces and interrupt management scheme. The board driver is also described in the KwikNet Device Driver Technical Reference Manual.
3. Using the KwikNet Configuration Builder, create and/or edit a Network Parameter File to select the KwikNet features which your application requires and to describe your network interfaces and their associated device drivers. On the Debug property page, enable some or all of KwikNet's debug features to assist you during initial testing. Use the builder to generate your KwikNet Library Make File.
4. Using the KwikNet Library Make File generated in step 3, create your KwikNet Library following the procedure to be described in Chapter 3.2.
5. Finally, create a make file which your make utility can use to build your application. It must compile your application modules, your KwikNet device drivers and your KwikNet board driver. It can then link the resulting object modules with your KwikNet Library, your RT/OS libraries and your C run-time library to create an executable load module. Follow the compilation and linking recommendations presented in Chapters 3.3, 3.4 and 3.5.
6. Use your software debugger and/or in-circuit emulator tools to transfer your load module to your target hardware. When testing, you should execute your application with a breakpoint on KwikNet procedure *kn_bphit()* so that you can readily detect fatal configuration or programming errors or unusual operation of the KwikNet TCP/IP Stack. Follow the testing guidelines presented in Chapter 1.9.

3.2 Making the KwikNet Library

To build the KwikNet Library, you will need a make utility capable of running your C compiler and object librarian (archiver). The library construction process is illustrated in Figure 3.2-1. If you have ported KwikNet to your operating environment, the shaded blocks indicate modules which you have already modified to adapt the make process to accommodate your software development tools. If you are using KwikNet with AMX, these modules are ready for use without modification.

Your custom KwikNet Library is created from the KwikNet Network Parameter File, a text file describing the TCP/IP features, options, networks and devices drivers which your application requires. This file is created and edited using the KwikNet Configuration Builder as described in Chapter 2.

The KwikNet Configuration Builder uses the information in your Network Parameter File to generate a KwikNet Library Make File. This make file is suitable for use with Microsoft's *NMAKE* utility. The make file purposely avoids constructs and directives that tend to vary among make utilities. Hence, you should have little difficulty using this make file with your own make utility if you so choose.

The make utility uses your C compiler and object librarian to generate the KwikNet Library from the KwikNet source modules and the OS Interface Module.

All KwikNet C files include a KwikNet **compiler configuration header file** *KNZZZCC.H*. This file identifies the characteristics of your C compiler. This file is also used to optimize code sequences within KwikNet modules by taking advantage of compiler specific features such as in-line code, assembly language functions and C library macros or functions. A number of variants of this module are provided with KwikNet ready for use with popular compilers on a variety of target processors.

The **OS Interface Module** *KN_OSIF.C* is the module which connects KwikNet to your RT/OS (see Figure 1.2-1 in Chapter 1). This module is merged into the KwikNet Library. The make process automatically includes the OS Interface Make File *KN_OSIF.INC* to determine the make dependencies and rules which control the compilation of the OS Interface source file *KN_OSIF.C*.

As you would probably expect, the make file does not know how to run your C compiler and object librarian. This information is provided in a file called *KNZZZCC.INC* which the make process automatically includes. This file, called a **tailoring file**, is used to tailor the library construction process to accommodate your make utility's syntax for implicit rules. It also provides the command sequences necessary to invoke your C compiler and object librarian. KwikNet is shipped with a number of tailoring files ready for use with Microsoft's *NMAKE* utility and many popular compilers.

Note

When KwikNet is used with AMX, the compiler configuration header file *KNZZZCC.H*, the OS Interface Make File *KN_OSIF.INC* and the tailoring file *KNZZZCC.INC* provided with KwikNet are ready for use without modification as described in Chapter 3.7.3.

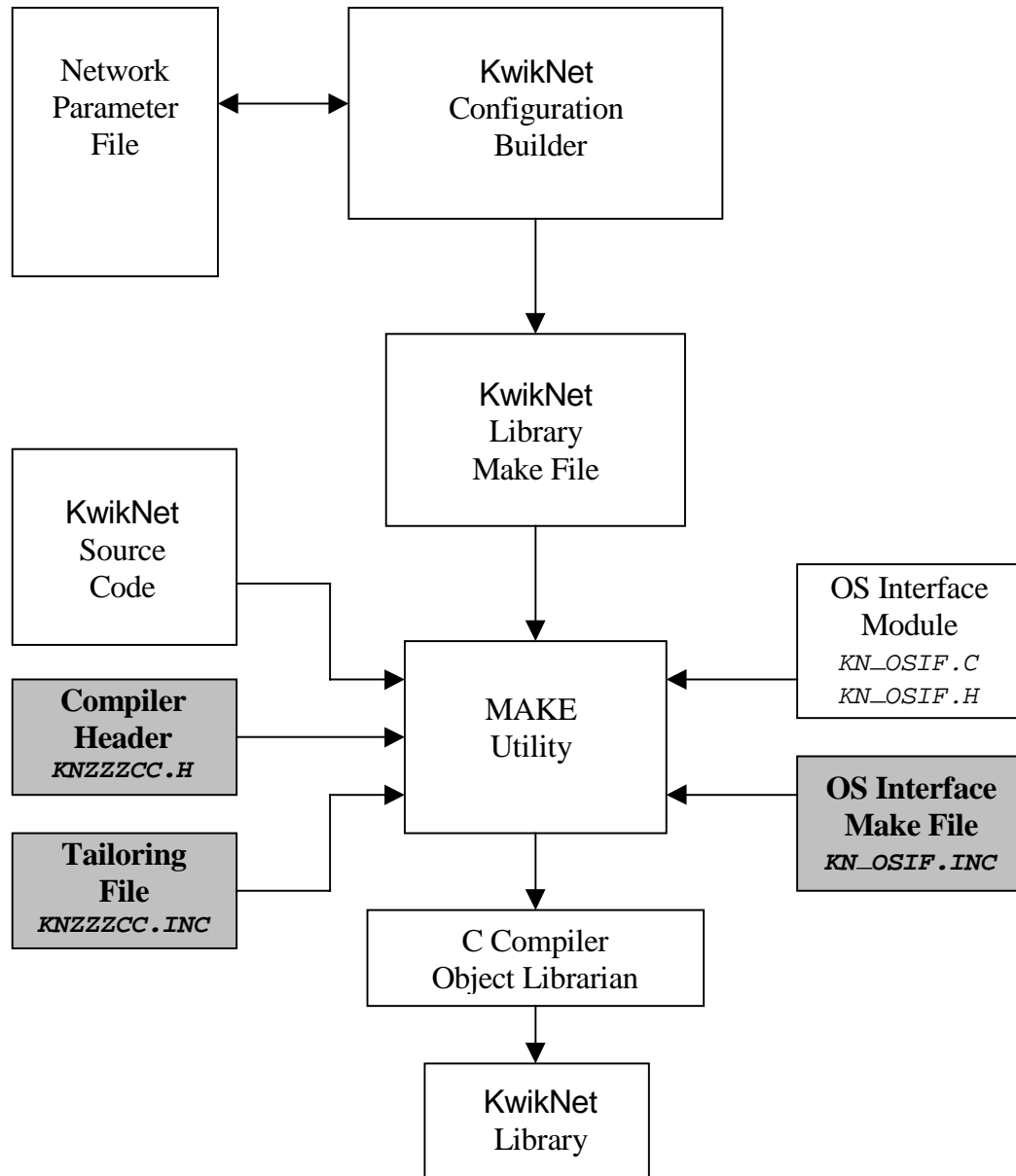


Figure 3.2-1 KwikNet Library Construction

KwikNet Directories and Files

The make process depends upon the structure of the KwikNet installation directory *KNTnnn*. When KwikNet is installed, the following subdirectories are created within directory *KNTnnn*.

<i>CFGBLDW</i>	KwikNet Configuration Builder; template files
<i>ERR</i>	Construction error summary
<i>MAKE</i>	KwikNet make directory
<i>TCPIP</i>	KwikNet header and source files
<i>TOOLXXX</i>	Toolset specific files
<i>TOOLXXX\LIB</i>	Toolset specific libraries will be built here
<i>TOOLXXX\DRIVERS</i>	KwikNet device drivers

Other directories containing sample programs will also be present but are not involved in the library construction process.

One or more toolset specific directories *TOOLXXX* will be present. There will be one such directory for each of the software development toolsets which KADAK supports. Each toolset vendor is identified by a unique two or three character mnemonic, *xxx*. The mnemonic *UU* identifies the toolset vendor used with the KwikNet Porting Kit.

Within directory *TOOLXXX* you will find a collection of files of the form *M_tttXXX.vvv*. These files, called **tailoring files**, are used to tailor the library construction process for the Microsoft make utility. The mnemonic *ttt* identifies the target processor. The extension *vvv* identifies the first version of the compiler for toolset *xxx* with which the tailoring file was tested. The tailoring file can be used with subsequent versions of the tools until some change in their method of operation requires a tailoring file update. For example, file *M_PPCMW.361* was first used to create the PowerPC KwikNet Library using Microsoft's *NMAKE* and the MetaWare v3.61 C compiler.

Getting Ready

Before creating the KwikNet Library, you must pick your C compiler and object module librarian (archiver). Be aware that KADAK has observed that not all compilers operate correctly with every version of the Microsoft make utility. If the make process inexplicably fails, it will most frequently be because of incompatibilities between these tools.

Pick the tailoring file which matches your choice of toolset and compiler version. Copy that file into the toolset directory but with name *KNZZZCC.INC*. You may have to overwrite the default copy created in that directory when KwikNet was installed.

KwikNet Library Make File

The KwikNet Configuration Builder is used to create and edit your Network Parameter File, say *NETCFG.UP*. It is this file which describes the KwikNet options and features which your application requires. From this parameter file, the Configuration Builder generates the KwikNet Library Make File, say *NETCFG.MAK*.

The Library Make File *NETCFG.MAK* is a make file which can be used to create the KwikNet Library tailored to your specifications. This make file is suitable for use with Microsoft's *NMAKE* utility.

Gathering Files

The block diagram in Figure 3.2-1 summarizes the components which are required to build the KwikNet Library. All of these files must be present in the appropriate KwikNet installation directories prior to making the KwikNet Library. Each of the following source files must be present in the indicated destination directory.

Source File	Destination Directory	File Purpose
<i>NETCFG.UP</i>	<i>MAKE</i>	KwikNet Network Parameter File
<i>NETCFG.MAK</i>	<i>MAKE</i>	KwikNet Library Make File
<i>KN_OSIF.C</i>	<i>TCPIP</i>	OS Interface Module
<i>KN_OSIF.H</i>	<i>TCPIP</i>	OS Interface Header File
<i>KN_OSIF.INC</i>	<i>TOOLXXX</i>	OS Interface Make Specification
<i>KNZZZCC.INC</i>	<i>TOOLXXX</i>	Tailoring File (for use with make utility)
<i>KNZZZCC.H</i>	<i>TOOLXXX</i>	Compiler Configuration Header File
<i>KNnnnnIP.LBM</i>	<i>TOOLXXX\LIB</i>	KwikNet Library Specification File

Creating the KwikNet Library

The KwikNet Library must be constructed from within directory *MAKE* in the KwikNet installation directory. Your Network Parameter File, say *NETCFG.UP*, and your Library Make File, say *NETCFG.MAK*, must be present in the KwikNet *MAKE* directory.

All of the compilers and librarians used at KADAK were tested on a Windows® workstation running Windows NT, 2000 and XP. However, you can build the library using any recent version of Windows, provided that your software development tools operate on that platform.

To create the KwikNet Library, proceed as follows. From the Windows Start menu, choose the MS-DOS Command Prompt from the Programs folder. Make the KwikNet installation *KNTnnn\MAKE* directory the current directory.

To use Microsoft's *NMAKE* utility, issue the following command.

```
NMAKE -fNETCFG.MAK "TOOLSET=XXX" "TRKPATH=treckpath"  
"OSPATH=yourospath" "KPF=NETCFG.UP" "AMX4ARM=ARM"
```

The make symbol *TOOLSET* is defined to be the toolset mnemonic *xxx* used by KADAK to identify the software tools which you are using.

The symbol *TRKPATH* is defined to be the string *treckpath*, the full path (or the path relative to directory *KNTnnn\MAKE*) to your Turbo Treck TCP/IP installation directory.

The symbol *OSPATH* is defined to be the string *yourospath*, the full path (or the path relative to directory *KNTnnn\MAKE*) to the directory containing your RT/OS components (header files, libraries and/or object modules). When using AMX, string *yourospath* is the path to your AMX installation directory.

The make symbol *KPF* is defined to identify the name of the Network Parameter File *NETCFG.UP* from which the KwikNet Library Make File *NETCFG.MAK* was generated. Both of these files must be present in the KwikNet *MAKE* directory.

For example, assume that the Turbo Treck TCP/IP release from Treck Inc. has been installed in directory *C:\TRECK* and that AMX 4-Thumb (part number PN422-1) has been installed in directory *C:\KADAK\AMX422*. Then, to build the KwikNet Library using Microsoft's *NMAKE* utility and Metrowerks tools, issue the following command.

```
NMAKE -fNETCFG.MAK "TOOLSET=ME" "TRKPATH=C:\TRECK"  
"OSPATH=C:\KADAK\AMX422" "KPF=NETCFG.UP"
```

Note that the extra make symbol definition string *"AMX4ARM=ARM"* is only appended to the make command line if you are using AMX 4-ARM (PN402-1).

By default, the KwikNet Library will be created in toolset dependent directory *TOOLXXX\LIB*. You can force the libraries to be created elsewhere by defining symbol *NETLIB=libpath* on the make command line. The string *libpath* is the full path (or the path relative to directory *KNTnnn\MAKE*) to the directory in which you wish the library to be created. You must copy the library specification file *KNnnnIP.LBM* from toolset *xxx* directory *TOOLXXX\LIB* to your alternate library directory *libpath*.

3.3 KwikNet Library Compilation Output

When you build the KwikNet Library as described in Chapter 3.2, the make utility will transfer files which are unique to KwikNet into the Treck installation directory. If KwikNet is installed in directory *C:\KNTnnn* and the Turbo Treck TCP/IP Stack is installed in directory *C:\TRECK*, then all KwikNet header and source files will be copied from directory *C:\KNTnnn\TCPIP* to the Treck installation directory. Header files will be copied to Treck directory *C:\TRECK\INCLUDE*. Source files will be copied to Treck directory *C:\TRECK\SOURCE*.

During the library construction process, all KwikNet source files and Turbo Treck TCP/IP source files will be compiled and the resulting object modules will be placed in directory *TOOLXXX\LIB*. The KwikNet Library will be created from these object files and placed in directory *TOOLXXX\LIB*. Note that the library file extension will be *.A* or *.LIB* or some other extension as dictated by the toolset which you are using.

<i>KNnnnIP.A</i>	KwikNet Library
------------------	-----------------

In addition to the library module and the object modules used to create them, the following files will also be created in directory *TOOLXXX\LIB*.

<i>KN_LIB.UP</i>	KwikNet Network Parameter File
<i>KN_LIB.MAK</i>	KwikNet Library Make File
<i>KN_LIB.H</i>	KwikNet Library Header File

File *KN_LIB.UP* is a copy of the Network Parameter File *NETCFG.UP* which you identified on your make command line. It is copied to the *LIB* directory so that you have a record of the parameters used to produce the library present in the directory.

File *KN_LIB.MAK* is the KwikNet Library Make File which can be used to reproduce the library. It is generated in the *LIB* directory so that you have a record of the make file used to produce the library present in the directory. This file is derived from the KwikNet Library Make Template file *KNnnnLIB.MT* and the parameters in Network Parameter File *KN_LIB.UP*. It should match the make file *NETCFG.MAK* which you passed to your make utility to start the make process.

File *KN_LIB.H* is the KwikNet Library Header File, a C header file generated by the make process. This file is derived from the KwikNet Library Header Template file *KNnnnLIB.HT* and the parameters in Network Parameter File *KN_LIB.UP*.

The KwikNet header file *KN_LIB.H* is also copied to Treck directory *C:\TRECK\INCLUDE* so that it is available for inclusion in the compilation of all C files in the library.

The toolset dependent header file *TOOLXXX\KNZZZCC.H* is also copied to Treck directory *C:\TRECK\INCLUDE* so that it is also available for inclusion in the compilation of all C files in the library.

3.4 Compiling Application Modules

In order to compile an application C source file, say *MYFILE.C*, which makes use of KwikNet services, the following KwikNet header files must be present in the Treck installation directory *C:\TRECK\INCLUDE*. These files will be copied there, if not already there, whenever a KwikNet Library is built. You must define the path to these header files using compiler switches or environment variables acceptable to your compiler.

<i>KN_LIB .H</i>	KwikNet Library Header File
<i>KN_API .H</i>	KwikNet Application Interface definitions
<i>KN_COMN .H</i>	KwikNet Common Interface definitions
<i>KN_OSIF .H</i>	KwikNet OS Interface definitions
<i>KNZZZCC .H</i>	KwikNet compiler specific definitions
<i>KN SOCK .H</i>	KwikNet Socket Interface definitions

Header file *KN_LIB.H* is a copy of your KwikNet Library Header File from your KwikNet library directory. This file is created as a byproduct of the KwikNet Library construction process described in Chapter 3.3.

Header files *KN_API.H*, *KN_COMN.H* and *KN_OSIF.H* are the KwikNet files with which all application modules must be compiled. Any module which includes *KN_LIB.H* will automatically include these header files as well. These files will be found in KwikNet installation directory *INCLUDE*.

Header file *KNZZZCC.H* is the compiler specific file which will be found in KwikNet installation directory *TOOLXXX*, where *XXX* is KADAK's mnemonic for a particular vendor's C tools.

Header file *KN SOCK.H* must be included by all applications which use TCP or UDP socket services.

If source file *MYFILE.C* makes calls to RT/OS service procedures, you must also have access to all of the required RT/OS header files.

You must also have access to your C library header files so that KwikNet header files can reference them.

With these header files in place, your application module *MYFILE.C* is ready to be compiled. If you are using KwikNet with AMX, the procedure is exactly as described in the toolset specific chapter of the AMX Tool Guide. If you have ported KwikNet using your own software development tools, the procedure will be the same as you used to compile the sample program source files.

Note

The make files for the sample programs provided with KwikNet and its optional components compile all of the modules which make up the sample program.

3.5 Linking the Application

To add KwikNet to your application, you must include the KwikNet Library module *KNnnnIP.A* in your link specification. Note that library file extensions may be *.A* or *.LIB* or some other extension dictated by the toolset which you are using.

You must also include the object module for your device driver, say *DDRIVER.O*, and your KwikNet board support module, say *KN_BOARD.O*, in your link specification. Note that object file extensions may be *.O* or *.OBJ* or some other extension dictated by the toolset which you are using.

The device driver object module(s) and the board support object module should be included together with all of your other application specific object modules. The KwikNet Library should precede the AMX or RT/OS libraries in the link sequence.

Although every effort has been made to ensure that each module in the KwikNet Library contains only forward references to other modules in the library, the goal has proved impossible to attain. Hence, some library modules do include backward references. This characteristic requires that the libraries be searched recursively until all resolvable references have been satisfied. Most linkers will meet this requirement. If yours does not, you will be forced to include the KwikNet Library more than once in your link specification.

If you are using KwikNet with AMX, there is little difference in linking an AMX application with or without KwikNet. Instructions for linking an AMX system are provided in the toolset specific chapters of the AMX Tool Guide.

3.6 Making the TCP/IP Sample Program

The KwikNet TCP/IP Stack includes a sample program, a working application that you can use to confirm the operation of KwikNet. Other sample programs are provided for use with optional KwikNet components such as the FTP Option and the Web Server.

The KwikNet Application Block Diagram (see Figure 1.2-1 in Chapter 1) summarizes the components which are fundamental to any KwikNet application. All of these components must be present in the appropriate KwikNet installation directories prior to making any of the KwikNet sample programs.

TCP/IP Sample Program Directories

When KwikNet is installed, the following subdirectories on which the TCP/IP Sample Program construction process depends are created within directory *KNTnnn*.

<i>CFGBLDW</i>	KwikNet Configuration Builder; template files
<i>ERR</i>	Construction error summary
<i>TCPIP</i>	KwikNet header and source files
<i>TOOLXXX</i>	Toolset specific files
<i>TOOLXXX\DRIVERS</i>	KwikNet device drivers and board driver
<i>TOOLXXX\LIB</i>	Toolset specific KwikNet Library will be built here
<i>TOOLXXX\SAM_MAKE</i>	Sample program make directory
<i>TOOLXXX\SAM_TCP</i>	KwikNet TCP/IP Sample Program directory
<i>TOOLXXX\SAM_COMN</i>	Common sample program source files

One or more toolset specific directories *TOOLXXX* will be present. There will be one such directory for each of the software development toolsets which KADAK supports. Each toolset vendor is identified by a unique two or three character mnemonic, *xxx*. The mnemonic *UU* identifies the toolset vendor used with the KwikNet Porting Kit.

Other subdirectories such as *TOOLXXX\SAM_FTP* or *TOOLXXX\SAM_WEB* will also be present for the sample programs provided with other optional KwikNet components.

TCP/IP Sample Program Files

To build the KwikNet TCP/IP Sample Program using make file *KNSAMPLE.MAK*, each of the following source files must be present in the indicated destination directory.

Source File	Destination Directory	File Purpose
.	CFGBLDW	KwikNet Configuration Builder; template files
	KwikNet source directory containing:	
KN_API.H	TCPIP	KwikNet Application Interface definitions
KN_COMN.H	TCPIP	KwikNet Common definitions
KN_OSIF.H	TCPIP	KwikNet OS Interface definitions
KN_SOCK.H	TCPIP	KwikNet Socket Interface definitions
	Toolset root directory containing:	
KN_OSIF.INC	TOOLXXX	OS Interface Make Specification
KNZZZCC.INC	TOOLXXX	Tailoring File (for use with make utility)
KNZZZCC.H	TOOLXXX	Compiler Configuration Header File
	KwikNet TCP/IP Sample Program directory containing:	
KNSAMPLE.MAK	TOOLXXX\SAM_TCP	TCP/IP Sample Program make file
KNSAMPLE.C	TOOLXXX\SAM_TCP	TCP/IP Sample Program
KNZZZAPP.H	TOOLXXX\SAM_TCP	TCP/IP Sample Program Application Header
KNSAMLIB.UP	TOOLXXX\SAM_TCP	Network Parameter File
KNSAMPLE.LKS	TOOLXXX\SAM_TCP	Link Specification File (toolset dependent)
		Other toolset dependent files may be present.
KNSAMSCF.UP	TOOLXXX\SAM_TCP	User Parameter File (for use with AMX)
KNSAMTCF.UP	TOOLXXX\SAM_TCP	Target Parameter File (for use with AMX)
	Common sample program source files:	
KNSAMOS.C	TOOLXXX\SAM_COMN	Application OS Interface
KNSAMOS.H	TOOLXXX\SAM_COMN	Application OS Interface header file
KNRECORD.C	TOOLXXX\SAM_COMN	Message recording services
KNCONSOL.C	TOOLXXX\SAM_COMN	Console driver
KNCONSOL.H	TOOLXXX\SAM_COMN	Console driver header
		Console driver serial I/O support:
KN8250S.C	TOOLXXX\SAM_COMN	INS8250 (NS16550) UART driver
KN_BOARD.C	TOOLXXX\DRIVERS	Board driver for target hardware
	KwikNet source directory containing additional header files for board, device, file and user application compilation:	
KN_ADMIN.H	TCPIP	KwikNet user administration interface definitions
KN_DVCIO.H	TCPIP	KwikNet common device I/O definitions
KN_FILES.H	TCPIP	KwikNet Universal File System (UFS) definitions
KN_VFS.H	TCPIP	KwikNet Virtual File System (VFS) definitions
KNFSUSER.H	TCPIP	KwikNet User File System definitions

TCP/IP Sample Program Parameter File

The Network Parameter File *KNSAMLIB.UP* describes the KwikNet options and features illustrated by the sample program. This file is used to construct the KwikNet Library for the TCP/IP Sample Program.

The Network Parameter File *KNSAMLIB.UP* also describes the network interfaces and the associated device drivers which the sample program needs to operate.

TCP/IP Sample Program KwikNet Library

Before you can construct the KwikNet TCP/IP Sample Program, you must first build the associated KwikNet Library.

Use the KwikNet Configuration Builder to edit the sample program Network Parameter File *KNSAMLIB.UP*. Use the Configuration Builder to generate the KwikNet Library Make File *KNSAMLIB.MAK*.

Look for any KwikNet Library Header File *KN_LIB.H* in your toolset library directory *TOOLXXX\LIB*. If the file exists, delete it to ensure that the KwikNet Library is rebuilt to match the needs of the TCP/IP Sample Program.

Then copy files *KNSAMLIB.UP* and *KNSAMLIB.MAK* into the KwikNet installation directory *KNTnnn\MAKE*. Use the Microsoft make utility and your C compiler and librarian to generate the KwikNet Library. Follow the guidelines presented in Chapter 3.2. The files listed in Chapter 3.3 will be generated by the make process.

Note

The KwikNet Library must be built before the TCP/IP Sample Program can be made. If file *KN_LIB.H* exists in your toolset library directory *TOOLXXX\LIB*, delete it to force the make process to rebuild the KwikNet Library.

The TCP/IP Sample Program Make Process

Each KwikNet sample program must be constructed from within the sample program directory in the KwikNet toolset directory. For example, the KwikNet TCP/IP Sample Program must be built in directory *TOOLXXX\SAM_TCP*.

All of the compilers and librarians used at KADAK were tested on a Windows® workstation running Windows NT, 2000 and XP. However, you can build each KwikNet sample program using any recent version of Windows, provided that your software development tools operate on that platform.

To create the KwikNet TCP/IP Sample Program, proceed as follows. From the Windows Start menu, choose the MS-DOS Command Prompt from the Programs folder. Make the KwikNet toolset *TOOLXXX\SAM_TCP* directory the current directory.

To use Microsoft's *NMAKE* utility, issue the following command.

```
NMAKE -fKNSAMPLE.MAK "TOOLSET=XXX" "TRKPATH=treckpath"  
"OSPATH=yourospath" "TPATH=toolpath"
```

The make symbol *TOOLSET* is defined to be the toolset mnemonic *xxx* used by KADAK to identify the software tools which you are using.

The symbol *TRKPATH* is defined to be the string *treckpath*, the full path (or the path relative to directory *TOOLXXX\SAM_TCP*) to your Turbo Trek TCP/IP installation directory.

The symbol *OSPATH* is defined to be the string *yourospath*, the full path (or the path relative to directory *TOOLXXX\SAM_TCP*) to the directory containing your RT/OS components (header files, libraries and/or object modules). When using AMX, string *yourospath* is the path to your AMX installation directory.

The symbol *TPATH* is defined to be the string *toolpath*, the full path to the directory in which your software development tools have been installed. For some toolsets, *TPATH* is not required. The symbol is only required if it is referenced in file *KNZZZCC.INC*.

The KwikNet TCP/IP Sample Program load module *KNSAMPLE.xxx* is created in toolset directory *TOOLXXX\SAM_TCP*. The file extension of the load module will be dictated by the toolset you are using. The extension, such as *OMF*, *ABS*, *EXE*, *EXP* or *HEX*, will match the definition of macro *XEXT* in the tailoring file *KNZZZCC.INC*.

The final step is to use your debugger to load and execute the KwikNet TCP/IP Sample Program load module *KNSAMPLE.xxx*.

3.7 Using KwikNet with AMX

3.7.1 AMX System Configuration

KwikNet includes its own interface to the underlying operating system. The KwikNet OS Interface for AMX is ready for use without modification or customization.

KwikNet makes few demands for AMX resources. Consequently, there are few changes to your AMX System Configuration Module required to accommodate KwikNet.

KwikNet Task

A single KwikNet Task drives the KwikNet TCP/IP Stack. You must add this task to your list of predefined tasks in your AMX System Configuration Module. You can use the AMX Configuration Builder to do so. The task's definition is as follows:

Tag	<i>KNET</i>
Procedure name	<i>kn_task</i>
Priority	Above all tasks which use KwikNet
Task stack size	AMX minimum plus 1024
Queue 0	0
Queue 1	0
Queue 2	0
Queue 3	0

With AMX 86 and some toolsets, you may have to add a leading or trailing underscore (`_`) character to the task procedure name. Note that the KwikNet Task must execute at a priority above all tasks which use KwikNet services.

The task stack size requirement will vary with the particular version of AMX you are using. As a good rule of thumb, choose a stack size which is approximately 1024 bytes more than the minimum stack size required for an AMX task. Add more stack if any of the following conditions exist.

- Target is a RISC processor with increased stack demands
- KwikNet options are used which make use of file system services
- KwikNet debugging aids are enabled in your KwikNet Library

For all versions of AMX, the KwikNet Task is a simple task with no AMX message queues. KwikNet uses private messaging blocks for internal communication with the KwikNet Task. The total number of available messaging blocks is defined by you on the Target property page when you create your KwikNet Network Parameter File. You may have to grow the number of messaging blocks if any of the following conditions exist.

- You have many tasks using network services
- You service several networks concurrently
- You expect high levels of network packet activity

AMX Interrupt Stack

You may have to grow the size of your AMX Interrupt Stack, the stack used by all Interrupt Service Procedures. The stack must be large enough to meet the needs of each of the KwikNet device drivers which service your physical network connections. As a good rule of thumb, choose a stack size which is approximately 500 bytes more than the minimum required AMX Interrupt Stack size.

KwikNet Semaphores

KwikNet requires one semaphore for its operation. This semaphore will be created dynamically by KwikNet during its initialization phase. You must include the AMX Semaphore Manager in your AMX System Configuration Module by declaring a requirement for at least one (1) semaphore. If you configure KwikNet to use standard C for memory allocation with memory locking enabled, an additional semaphore will be needed. If you use a file system other than AMX/FS and require file access locking, allocate one more semaphore. Hence, the base requirement is for 1, 2 or 3 semaphores.

To the base requirement, you must add one semaphore for each application task that can access KwikNet. For example, if your base requirement is 2 (one for KwikNet and one for memory locking using standard C) and you have three tasks that concurrently use KwikNet services, you must provide at least five semaphores for the KwikNet stack, over and above any that you may require for your application.

KwikNet Memory Pool

If you configure KwikNet to use AMX memory management services, one AMX memory pool will be required. The memory pool will be created dynamically by KwikNet during its initialization phase. You must include the AMX Memory Manager in your AMX System Configuration Module by declaring a requirement for at least one memory pool. The memory for the pool must be allocated by you. Use the KwikNet Configuration Builder to edit your Network Parameter File and select one of the memory assignment techniques specified on the OS property page.

KwikNet Timer

KwikNet requires one AMX timer for its operation. This timer will be created dynamically by KwikNet during its initialization phase. You must include the AMX Timer Manager in your AMX System Configuration Module by declaring a requirement for at least one timer. Of course, to support timing you must also include an AMX Clock Handler as part of your application.

The KwikNet timer operates at the network clock frequency defined by you in your KwikNet Network Parameter File. The period of the network clock must correspond to an integer multiple of AMX system ticks. For example, you may have a hardware clock interrupt frequency of 1KHz with an AMX tick frequency of 100 Hz and a KwikNet network frequency of 10 Hz. In this case, the KwikNet timer will operate at 100 ms intervals measured with 10 ms resolution.

KwikNet Restart and Exit Procedures

You must include KwikNet Restart Procedure *kn_osready* first (or near first) in your list of Restart Procedures in your AMX System Configuration Module.

KwikNet includes a startup procedure *kn_enter* and a shutdown procedure *kn_exit*. You can include the KwikNet startup procedure *kn_enter* in your list of Restart Procedures in your AMX System Configuration Module. It is this procedure which starts the KwikNet Task to initialize the KwikNet TCP/IP Stack. Alternatively, one of your own Restart Procedures can call *kn_enter()*. The position of this procedure in the list of Restart Procedures is not critical since no KwikNet services can be used by tasks until the KwikNet Task has executed. Another approach is to have a task call *kn_enter()* to start KwikNet. Be certain that no other task tries to use KwikNet services until KwikNet is operational.

If your AMX application allows an orderly shutdown and exit from AMX, you can add the KwikNet shutdown procedure *kn_exit* to your list of Exit Procedures in your AMX System Configuration Module. Alternatively, one of your own Exit Procedures can call *kn_exit()*. Insert this procedure into the list at the point in the exit sequence at which the KwikNet TCP/IP Stack is no longer required. You must ensure that all tasks have stopped using KwikNet services before you allow KwikNet to shut down. For KwikNet to shut down, you must use the KwikNet Configuration Builder to edit your Network Parameter File and enable KwikNet shutdown on the General property page.

AMX 86 PC Supervisor

AMX 86 includes a component called the PC Supervisor which permits AMX 86 to be used with DOS on PC platforms. Special care must be taken when using the PC Supervisor with AMX 86 and KwikNet.

The PC Supervisor's Clock Tick Task and Keyboard Task must be of higher priority than the KwikNet Task to ensure that they operate without interference from network activity.

All tasks which use KwikNet services must be of lower priority than the KwikNet Task. The PC Supervisor Task must be of lower priority than all application tasks which use KwikNet so that it does not interfere with their use of the network.

These task prioritization rules work provided that tasks which use KwikNet services never go compute bound. For example, if a task continuously polls KwikNet to test for the completion of some network operation, then any higher priority task which attempts to use DOS services will appear to hang because the low priority PC Supervisor Task is unable to execute to service the DOS request. In such cases, you will have no choice but to raise the priority of the PC Supervisor Task and accept the fact that DOS operations can temporarily block tasks of lower priority.

If you examine the KwikNet Sample Program provided with AMX 86, you will observe that the PC Supervisor Task has actually been placed at a priority above the KwikNet Task. This violation of the priority rules was done intentionally for the following reason. The Sample Program can operate without any physical network interfaces. Consequently, the application tasks can execute in a compute bound fashion because they never have to wait for real devices to respond. This scenario prevents the PC Supervisor Task from servicing any request by the Sample Program's Print Task to present messages on the PC display screen. By raising the priority of the PC Supervisor Task above that of the KwikNet Task, all messages appear on the PC display screen as soon as they are generated, making it easier for you to observe the actual sequence of operations.

3.7.2 AMX Target Configuration

Each KwikNet device driver for AMX includes an Interrupt Service Procedure consisting of two (sometimes three) parts. All drivers require an ISP root and an Interrupt Handler. Some versions of AMX also require the driver to provide an ISP stem.

An ISP root is required for each device interrupt source which the KwikNet board driver module *KN_BOARD.C* is configured to support. Unless modified by you, the board driver supports four ISP roots with names of the form *kn_isprootX()* (*X* is *A*, *B*, *C* or *D*). KwikNet dynamically assigns each network device to one of these ISP roots when the network is initialized.

32-Bit AMX Systems

Each ISP root is serviced by a common Interrupt Handler *kn_isphandler()* located in the KwikNet board driver module *KN_BOARD.C*. The handler is called with a single pointer parameter which identifies the network device which generated the interrupt. Four parameters with names of the form *kn_ispparamX* (*X* is *A*, *B*, *C* or *D*) are provided in the board driver module, one for each of the four ISP roots with like names.

An ISP stem *kn_ispstem()* in the KwikNet board driver module *KN_BOARD.C* is provided when required by AMX. The ISP stem also receives the device specific parameter *kn_ispparamX* from the ISP root.

For all 32-bit implementations of AMX, the ISP must be described in the AMX Target Configuration Module. You can use the AMX Configuration Builder, a Windows[®] utility, to edit the Target Parameter File to add ISP definitions.

There must be one ISP definition for each of the device interrupt sources which the KwikNet board driver module *KN_BOARD.C* is configured to support. Each ISP definition identifies the names of the ISP root, the ISP Handler and the ISP stem if applicable. Each ISP definition also provides the appropriate pointer parameter *kn_ispparamX*. No interrupt vector is included in the definition since each KwikNet device driver automatically installs the pointer to its ISP root into the AMX Vector Table when KwikNet is initialized by the KwikNet Task.

See the KwikNet TCP/IP Sample Program Target Parameter File *KNSAMTCF.UP* for an example of the definition of the four ISPs supported by the KwikNet board driver module *KN_BOARD.C*.

16-Bit AMX 86 Systems

AMX 86 does not utilize a Target Configuration Module. The KwikNet board driver provided with AMX 86 creates an ISP root named *kn_isprootX()* (*X* is *A*, *B*, *C* or *D*) for each of the device interrupt sources which it is configured to support. Each ISP root *kn_isprootX()* calls its Interrupt Handler *kn_ispsrcX()* which in turn calls procedure *kn_isphandler()* with the device specific *kn_ispparamX* parameter. All of these procedures are located in the board driver module *KN_BOARD.C*. Each KwikNet device driver for AMX 86 automatically installs the pointer to one of these ISP roots into the AMX Vector Table when KwikNet is initialized by the KwikNet Task.

3.7.3 Toolset Considerations

Tailoring Files

The KwikNet Library is constructed using your make utility, C compiler and object module librarian. A file which KADAK calls a **tailoring file** is used to tailor the library construction process for a particular C compiler and object librarian. Separate tailoring files are available for each toolset combination which KADAK supports. These tailoring files are provided ready for use with Microsoft's *NMAKE* utility.

KADAK uses a 2 or 3 character toolset mnemonic to identify each supported toolset combination. The tailoring files for toolset *xxx* are located in directory *TOOLxxx* in the KwikNet installation directory *KNtnnn*. Use tailoring files *M_tttxxx.vvv* with Microsoft's *NMAKE*. The mnemonic *ttt* identifies the target processor. The extension *vvv* identifies the first version of the compiler for toolset *xxx* with which the tailoring file was tested. The tailoring file can be used with subsequent versions of the tools until some change in their method of operation requires a tailoring file update. For example, file *M_PPCDA.42* was first used to create the PowerPC KwikNet Library using Microsoft's *NMAKE* and the Diab Data (toolset *DA*) v4.2 (42) C compiler.

Note

Pick the tailoring file which matches your choice of toolset and compiler version. Copy that file into toolset directory *TOOLxxx* but with name *KNZZZCC.INC*. You may have to overwrite the default copy created when KwikNet was installed.

Compiler Configuration Header File

All KwikNet C files include a KwikNet compiler configuration header file *KNZZZCC.H*. This file identifies the characteristics of your C compiler. When KwikNet is used with AMX, the compiler configuration header file *KNZZZCC.H* installed in KwikNet directory *TOOLxxx* is ready for use with the C compiler for toolset *xxx* without modification.

OS Interface Make File

The OS Interface Module *KN_OSIF.C* is the module which connects KwikNet to AMX (see Figure 1.2-1 in Chapter 1). This module is merged into the KwikNet Library. The make process automatically includes the OS Interface Make File *KN_OSIF.INC* to determine the make dependencies and rules which control the compilation of the OS Interface source file *KN_OSIF.C*. When KwikNet is used with AMX, the OS Interface Make File *KN_OSIF.INC* is installed in KwikNet directory *TOOLxxx* is ready for use with toolset *xxx* without modification.

3.7.4 AMX Application Construction Summary

Construction of any KwikNet application for use with AMX will closely follow the steps needed to build the KwikNet TCP/IP Sample Program. These steps are summarized below. Note that the make file provided with KwikNet sample programs actually does steps 5, 6, 7 and 8.

1. Using the KwikNet Configuration Builder, open the Sample Program's Network Parameter File *KNSAMLIB.UP* (see Chapter 2.3). Use the builder to generate your KwikNet Library Make File *KNSAMLIB.MAK*. Use this file, the Microsoft make utility and your C compiler and librarian to generate the KwikNet Library (see Chapter 3.2).
2. If you wish to use your own clock driver, do step 3. If you wish to use your own serial driver for logging messages to a terminal, do step 4. Otherwise, go to step 5.
3. If you wish to use your own working AMX Clock Driver instead of the simulated clock provided by the sample program, edit the Sample Program User Parameter File and Target Parameter File (not required for AMX 86) to accommodate your clock driver. Edit the KwikNet Sample Program Link Specification File *KNSAMPLE.LKS* to include your clock driver object module.
4. If you ported the AMX Sample Program serial I/O driver to your hardware and wish to use it to log messages to a terminal, edit the sample program Application Header file *KNZZZAPP.H* and define symbol *KN_CS_DEVTYPE* to be *KN_CS_DEVAMX*. Edit the KwikNet Sample Program Link Specification File *KNSAMPLE.LKS* to include your serial driver object module.
5. Using the AMX Configuration Builder, open the Sample Program's User Parameter File *KNSAMSCF.UP*. Use the builder to generate the AMX System Configuration Module *KNSAMSCF.C*. Compile the module as described in the AMX Tool Guide for the toolset which you are using.
6. If you are using AMX 86, go to step 7. Otherwise, using the AMX Configuration Builder, open the Sample Program's Target Parameter File. Use the builder to generate the AMX Target Configuration Module. Assemble the module as described in the AMX Tool Guide for the toolset which you are using.
7. Compile the KwikNet TCP/IP Sample Program application modules listed in Chapter 3.6. Compile these modules with full debug information to improve your view when running the sample with your debugger.
8. Link the modules listed in the KwikNet Sample Program Link Specification File *KNSAMPLE.LKS* together with your C Library to create your KwikNet application load module (see Chapter 3.5).
9. Use your debugger to load and execute the KwikNet Sample Program.

This page left blank intentionally.

4. KwikNet Low Level Services

4.1 The UDP Programming Interface

Applications which are memory constrained or have no need to use TCP can exclude the TCP protocol and use only the KwikNet IP stack and its UDP application programming interface (API). Be aware that UDP is a connectionless protocol which uses only the unreliable IP layer for UDP datagram delivery.

There are two ways to access UDP services: use UDP sockets or use UDP channels. You can program using UDP sockets or UDP channels or a mixture of both if you so choose.

UDP Sockets

KwikNet supports the use of the UDP protocol with the same sockets interface used with the TCP/IP protocol (see Chapter 5). The use of UDP sockets is the preferred method of UDP communication. It is the only method which supports both IPv4 and IPv6.

UDP Channels

The UDP channel is an artifice first introduced by KADAK with KwikNet v2 to support UDP programming. For backwards compatibility with KwikNet v2, UDP channels are supported in all versions of KwikNet up to and including KwikNet v3.01a. However, future releases of KwikNet will no longer support this feature.

UDP channels can only be used with IPv4. UDP channels cannot be used with IPv6.

Note

UDP channels cannot be used with IPv6.

Support for UDP channels will not be provided in future releases of KwikNet.

The UDP Channel

KwikNet offers an alternate, low level UDP API which you enable by editing your Network Parameter File and checking the box labeled "UDP channel support" on the IPv4 property page (see Chapter 2.3).

KwikNet defines an abstraction called a UDP channel which can be used to control the sending and receiving of UDP datagrams on the network. UDP datagrams cannot be sent without first acquiring a UDP channel. Received UDP datagrams are rejected if an associated UDP channel does not exist.

To send or receive UDP datagrams, you must first call KwikNet procedure *kn_udpopen()* to open a UDP channel. In the call you must provide the IP address of the foreign host with whom you wish to correspond. An IP address of 0.0.0.0 is used to indicate that you will accept UDP datagrams from any foreign host. If you will only accept UDP datagrams from a specific foreign host, you must also provide the protocol port number for the foreign host. A foreign port number of 0 can be used to indicate that you will accept a UDP datagram from any port at that foreign host.

The KwikNet procedure *kn_udpopen()* returns a handle which uniquely identifies the UDP channel allocated by KwikNet for your use. This handle can be used in calls to *kn_udpsend()* to send data through the UDP channel to *any* foreign host. However, you will only be able to receive UDP datagrams from the foreign host identified when the UDP channel was opened.

When you open the UDP channel, you can also bind the channel to a specific local IP address and/or port. If you do provide an IP address but no port number, a port number will be assigned for you. In this case, the foreign host will not be able to identify your port until you send a datagram to the foreign host.

If you open the UDP channel with no local IP address or port specified, you will have to send a datagram to the foreign host before it will be able to communicate with your local host. Alternatively, you can use KwikNet procedure *kn_udpbind()* to bind your UDP channel to a specific local IP address and port at some point after you have opened the UDP channel but before you have sent any datagram to the foreign host. Having done so, you can receive UDP datagrams on your UDP channel without first having to send a UDP datagram to the foreign host to identify your IP address. Of course, that presumes that the foreign host knows the IP address and port to which you are bound.

Once your application is finished conversing with the foreign host, it must call KwikNet procedure *kn_udpclose()* to close the UDP channel. The handle used to access the UDP channel becomes invalid once the channel is closed.

Receiving UDP Datagrams

If you expect to receive a UDP datagram from a foreign host, your open request must provide a pointer to an application callback function which KwikNet can call upon receipt of such a UDP datagram. The callback function is prototyped as follows:

```
int user_udprecv(struct knx_udpmsg *msgp, void *userp);
```

Parameter *userp* is an application pointer provided by you in your request to open the channel on which this UDP datagram was received. It is a copy of the parameter found at *msgp->xudpm_user*.

Parameter *msgp* is a pointer to a KwikNet UDP message descriptor. Structure *knx_udpmsg* is defined in KwikNet header file *KN_API.H* as follows:

```
struct knx_udpmsg {
    struct in_addr xudpm_src;          /* IP address of source */
    struct in_addr xudpm_dest;         /* IP address of destination */
    int xudpm_fport;                   /* Foreign port (source) */
    int xudpm_lport;                   /* Local port (destination) */
    char *xudpm_datap;                 /* Pointer to UDP data */
    int xudpm_length;                  /* Length of UDP data */
    int xudpm_rsv1;                    /* Reserved for alignment */
    void *xudpm_user;                  /* User parameter */
    unsigned long xudpm_channel;        /* UDP channel ID */
    struct knx_lmnode xudpm_node;       /* List node (for use by user) */
    void *xudpm_handle;                /* Reserved for KwikNet use */
};
```

The UDP message structure describes the received UDP datagram. Fields *xudpm_src*, *xudpm_dest*, *xudpm_fport*, and *xudpm_lport* are extracted from the received packet and presented to your application in an easy to use form. The source and destination IPv4 addresses are presented in net endian form in field *s_addr* of structure *in_addr*. The foreign and local ports are provided as integers in host endian form.

The data within the UDP datagram is located at the memory address specified by field *xudpm_datap*. The length of the data region is specified by field *xudpm_length*.

Your UDP callback function must return 0 if it accepts the UDP message descriptor. In this case, once your application has finished processing the UDP datagram, it must call KwikNet procedure *kn_udpfree()* to release the UDP message descriptor and free the associated data storage for reuse by KwikNet.

Your UDP callback function must return -1 if it cannot accept the message descriptor. In this case, KwikNet will release the UDP message descriptor and free the associated data storage. It is important to note that KwikNet will not send an ICMP destination unreachable message to the originator of the rejected UDP datagram.

Processing Received UDP Datagrams

Your UDP callback function executes in the context of the KwikNet Task. Your function must not initiate any operation which would force the KwikNet Task to be blocked waiting for some event.

In most multitasking applications, it is recommended that your UDP callback function pass the UDP message descriptor to some other application task for processing. Often that task will be the same task that opened the UDP channel and initiated the conversation in the first place. Field *xudpm_node* in the UDP message descriptor is a KwikNet list node that your application can use to attach the descriptor to a list managed using the private KwikNet List Manager services (see procedures *kn_lmxxxxx()* in source file *KN_LMGR.C*).

Even in single threaded systems, your UDP callback function should pass the UDP message descriptor to your App-Task for processing.

Note that your task has access to the copy of your application parameter located in field *xudpm_user* in the UDP message descriptor. Note that field *xudpm_channel* contains the UDP channel ID, the user handle acquired when you called *kn_udpopen()*.

When your task finishes processing the UDP datagram, it must call KwikNet procedure *kn_udpfree()* to release the UDP message descriptor and free the associated data storage for reuse by KwikNet.

Broadcast UDP Datagrams

A broadcast UDP datagram is a message directed to IP address 255.255.255.255. You can send and receive broadcast UDP datagrams on a UDP channel. When a broadcast UDP datagram is received, it is delivered to the first UDP channel which KwikNet can find with a matching local port number.

UDP Echo Requests

A UDP datagram directed to well known port 7 is called a UDP Echo Request. Your application can handle UDP echo requests by opening a UDP channel on port 7. Any UDP datagram received for port 7 will be passed to your UDP callback function.

4.2 DHCP, BOOTP and Auto IP

4.2.1 DHCP and BOOTP

KwikNet includes support for the Dynamic Host Configuration Protocol (DHCP) which permits an Ethernet network's IP address to be dynamically assigned when the network is first started. The DHCP Client which implements this service is provided as a standard feature with systems that only incorporate IPv4. For dual IPv4/IPv6 systems requiring a DHCP Client, an alternate, optional KwikNet DHCP component is available.

DHCP support is enabled by a configuration parameter in the KwikNet Network Parameter File used in the construction of the KwikNet Library as described in Chapter 2.3. When you define your KwikNet configuration, simply check the option box labeled "DHCP Client support" on the IPv4 property page.

Once you have enabled DHCP support, KwikNet allows each Ethernet network interface to be individually DHCP enabled. For a prebuilt network interface, edit your Network Parameter File, go to the Networks property page and add or edit your Ethernet network interface description. Click on the Edit: IP Address radio button and select DHCP, DHCP (reboot) or BOOTP from the pull down list labeled "Runtime configuration:" (see Chapter 2.4).

For network interfaces that are dynamically created at runtime, your application must request DHCP support when function *kn_ifopen()* is called to open the interface.

The KwikNet DHCP Client supports three variants: standard DHCP, DHCP with reboot and BOOTP. The DHCP options operate identically with one exception. If the reboot variant is used, the DHCP client will always start the negotiation in the reboot state and demand that the DHCP server grant the use of the configured IP address. The BOOTP option uses the older BOOT Protocol to acquire an IP address.

Note

KwikNet does not provide DHCP or BOOTP support for SLIP or PPP networks.

DHCP and BOOTP Operation

The KwikNet DHCP Client automatically requests an IP address for a DHCP enabled Ethernet network whenever the network interface is opened.

The DHCP client uses UDP datagrams for the transmission of DHCP messages. Responses are expected to be in UDP datagrams.

The DHCP client broadcasts a DHCP query to all DHCP servers. If the network has been configured for standard DHCP operation and has been configured with an IP address other than 0.0.0.0, the DHCP client will request the use of that IP address but accept an alternate if offered by the DHCP server.

If the network has been configured for DHCP (reboot) operation, it must be configured with an IP address other than 0.0.0.0. In this case, the DHCP client will demand the use of that IP address and will accept no other. Note that the broadcast DHCP query looks like a valid BOOTP query to any server which only supports the older BOOTP format.

Once an IP address offer is received from a DHCP server, the DHCP client responds with a request to unconditionally accept the offer. If the DHCP server acknowledges the acceptance of its offer, the DHCP client adopts the IP address thereby making the network ready for use by the application.

If a BOOTP server responds to the initial DHCP query, the DHCP client simply adopts the IP address provided by the BOOTP server thereby making the network ready for use by the application. There is no need for the acceptance and acknowledgment handshake.

The KwikNet DHCP Client does not make use of the server host name or boot file name, if any, provided in the DHCP or BOOTP server response. To access this information, call Treck function `tfConfGetBootEntry()`.

DHCP Timeout

If no response is received from any DHCP or BOOTP server within the timeout interval (initially four seconds), the DHCP client resends its broadcast query and increases its timeout interval by a factor of 2^n where n is the number of failed attempts thus far. The retry timeout value is not allowed to exceed an upper limit of 64 seconds. If an IP address cannot be acquired, the DHCP client keeps retrying for approximately three minutes after which the cycle is repeated. This process continues forever if an IP address cannot be found.

DHCP Leases

When a DHCP server provides an IP address, it grants the network interface a lease to use that address for a specific interval. The DHCP client always requests a permanent lease but can live with a limited lease interval if that is all that is granted by the DHCP server. Note that the lease granted by a BOOTP response is considered to be permanent.

The DHCP client always tries to renew a limited time lease by negotiating with the DHCP server which granted the lease. The DHCP server dictates when the lease will expire and can specify the points in time at which the lease renewal should be attempted. By default, in the absence of contrary directions from the DHCP server, the DHCP client will attempt to renew its lease as follows. If the lease interval is L seconds, the DHCP client begins a lease renewal negotiation after $L/2$ seconds. Negotiation requests are repeated at intervals until $7L/8$ seconds into the lease. Each interval is half the period from the time of the last request to the $7L/8$ seconds mark, but never less than 60 seconds.

If a new lease is not granted, the DHCP client will initiate a new negotiation sequence by broadcasting a query to all DHCP servers requesting the use of the expired IP address. If the request is not granted by any server, the DHCP client starts over, trying once again to acquire the initially configured IP address, if any, from any DHCP server.

DNS Server Identification Via DHCP

When a DHCP server responds to an IP address query, it can also provide a list of IP addresses for known Domain Name System (DNS) servers. The DHCP client accepts up to two DNS server IP addresses. The first member of the list is deemed the primary DNS server; the second is the secondary DNS server. The server addresses are recorded and can be accessed by your application for presentation to your KwikNet DNS Client, if it exists. To access the DNS server information, call Treck function `tfConfGetBootEntry()`.

4.2.2 Auto IP Operation

There is an optional KwikNet component which provides the Auto IP service which an Ethernet network interface can use to assign its own IP address dynamically when the network is first started. This feature is enabled by a configuration parameter in the KwikNet Network Parameter File used in the construction of the KwikNet Library as described in Chapter 2.3. When you define your KwikNet configuration, simply check the box labeled "Auto IP address discovery" on the IPv4 property page.

Once you have enabled Auto IP support, KwikNet allows each Ethernet network interface to be individually Auto IP enabled. To do so, edit your Network Parameter File, go to the Networks property page and add or edit your Ethernet network interface description. Click on the Edit: IP Address radio button and select Auto IP from the pull down list labeled "Runtime configuration:" (see Chapter 2.4).

Whenever a network interface which is configured to use Auto IP is opened, the network driver begins an Ethernet ARPing process to claim ownership of an IP address in the range 169.254.1.0 to 169.254.254.255. It makes a random guess at an IP address in this range and uses ARP requests with collision detection enabled to determine if the address is already in use. The process continues indefinitely until an unused IP address can be detected and assigned to the network interface. At that time the network interface is declared operational.

Note

KwikNet does not provide Auto IP support for SLIP or PPP networks.

4.2.3 IP Address Notification

A network which uses DHCP, BOOTP or Auto IP to acquire an IP address is not usable until such time as the IP address becomes available. At that time the network interface is declared up and available for service. Your application can periodically call KwikNet procedure *kn_ifstate()* to determine when the network interface is first available.

Alternatively, you can provide a network event notification function *kn_netevent()* which will be called whenever any network state transition or other significant network event occurs. To enable this feature, edit your KwikNet Network Parameter File and check the option box labeled "Enable network event notification" on the General property page.

Function *kn_netevent()* will be called whenever the DHCP client fails to acquire an IP address, successfully negotiates an IP address or loses its lease for a previously granted IP address.

Function *kn_netevent()* will also be called whenever the Auto IP service fails or succeeds to acquire an IP address for a network interface.

4.3 The DNS Client

The KwikNet DNS Client is an optional component providing support for the Domain Name System (DNS) which permits a network's IP address to be derived from a name string. For example, the IP address for KADAK's Internet website can be derived by doing a DNS query for the name string `www.kadak.com` using Treck function `tfDnsGetHostByName()`.

This feature is enabled by setting the DNS configuration parameters in the KwikNet Network Parameter File used in the construction of the KwikNet Library as described in Chapter 2.3. This service is provided by the KwikNet DNS Client which is configured to operate as specified by you in your KwikNet configuration. The DNS client executes in response to queries from your application and then relies upon the KwikNet Task to resolve the query, if necessary.

DNS Servers

The KwikNet DNS Client maintains the IP addresses of two DNS servers called the primary and secondary DNS servers. When KwikNet starts, the DNS client clears both DNS server IP addresses. Until your application provides a primary and optional secondary DNS server, any DNS query will fail.

The KwikNet DHCP Client accepts up to two DNS server IP addresses for each Ethernet network interface configured to support DHCP. You can access these DNS server addresses by calling Treck function `tfConfGetBootEntry()`.

The KwikNet PPP network driver accepts up to two DNS server IP addresses for each PPP network interface configured to accept DNS server addresses. You can access these DNS server addresses by calling Treck function `tfGetPppDnsIpAddress()`.

Given these DNS server addresses or your own derived from some other source, you can set the DNS client's primary or secondary DNS address by calling Treck function `tfDnsSetServer()`. To remove a previously installed DNS server IP address, call Treck function `tfDnsSetServer()` with a primary or secondary IP address of 0.0.0.0.

DNS Queries

The DNS client will always start a new DNS name lookup by querying the primary DNS server first. If no primary DNS server exists, the query will fail. If the query to the primary DNS server fails, the DNS client will then query the secondary DNS server, if one exists.

The results of each DNS query are kept in a name cache maintained by the DNS client. Each cached entry includes one or more of the IP addresses provided by the DNS server which resolved the name. The total number of cached names is determined by you when you configure your KwikNet Library.

Once the KwikNet TCP/IP Stack has been initialized, your application can call Treck function `tfDnsGetHostByName()` to make a DNS name query. You can also call Treck function `tfDnsGetHostAddr()` to find the DNS name for a specific IP address, a form of reverse DNS lookup.

In a multitasking system, any task making such a query must be of lower priority than the KwikNet Task. If the name is already in the DNS client's name cache, you will immediately be given the first available IP address from the list in that name's cache entry. If a new query must be made and there is no room in the name cache, the oldest cached name will be purged so that your request can be granted. The KwikNet DNS Client will then initiate the query.

The Treck functions return with an error code indicating the success or failure of the operation. If you have configured the DNS client to operate in non-blocking mode, the Treck functions may return with error code `TM_EWOULDBLOCK` indicating that your query is underway. In this case, you must repeat the query periodically to await the final result of the query, indicated by a return value other than `TM_EWOULDBLOCK`.

If you have configured the DNS client to operate in blocking mode, the Treck functions will only return when the query is complete. The return value will indicate the success or failure of the operation.

Get Host By Name

Many networking systems provide the function `gethostbyname()` which finds the IP address for a host with a specific domain name. The prototype for this function indicates that the function is inherently non-reentrant. For this reason, KwikNet provides the alternate, reentrant Treck function `tfDnsGetHostByName()` better suited for use in multitasking systems.

4.4 ICMP Protocol

KwikNet includes support for the subset of Internet Control Message Protocol (ICMP) services it needs for proper operation of any of the protocols which can be utilized with KwikNet and are dependent on ICMP.

KwikNet does not issue or reply to timestamp or information requests.

KwikNet supports ICMP destination unreachable datagrams. If such a datagram is received, KwikNet updates its routing tables to reflect the fact that the specified destination has been declared unreachable through a particular network interface. If KwikNet discards an IP datagram because it cannot be handled properly, an ICMP destination unreachable message is sent to the host from which the rejected IP datagram was received.

If you are using the KwikNet SNMP Agent, KwikNet maintains counts of the various ICMP datagrams which it sends and receives. These ICMP statistics can be accessed within the MIBs managed by the SNMP Agent.

ICMP and Raw Sockets

Your application can use the Treck raw sockets API to read ICMP echo requests and ICMP address mask requests. Your application will receive a duplicate of the received packet. You must not issue a reply because KwikNet will have already done so.

This feature is enabled by a configuration parameter in the KwikNet Network Parameter File used in the construction of the KwikNet Library as described in Chapter 2.3. When you define your KwikNet configuration, simply check the option box labeled "Receive ICMP datagrams via raw sockets" in the Sockets region of the TCP property page.

Using PING

KwikNet always replies to an ICMP echo request (a PING) with an ICMP echo reply. You can use the Treck raw socket API to read these PING requests.

Treck provides a separate PING service which you can use to initiate the sending of periodic PINGs (ICMP echo request datagrams) to a specific foreign host. You access this service using the PING API described in Chapter 6 of the Treck TCP/IP User Manual.

This page left blank intentionally.

4.5 KwikNet Network Interface Services

Introduction to Network Interfaces

Most embedded systems include one or two network interfaces to connect the system to external networks. For such systems, the simplest approach to incorporating the interface into your application is to use the KwikNet Configuration Builder to add the network interface to your KwikNet Library. KwikNet will then automatically add the network interface to its list of available networks and initialize the interface as soon as KwikNet is started. Each of these prebuilt networks is opened and made ready for use by your application.

In some cases, there may be a need for the application to control the interface startup process. To meet this requirement, KwikNet provides a collection of network interface management services which you can use to dynamically add, open and close networks at runtime. These services also allow you to control the operation of your prebuilt networks. By dynamically adding your network interface, you can also configure it to make use of advanced features offered by the Turbo Treck TCP/IP Stack but not automatically available with KwikNet's prebuilt networks. All KwikNet network interface management services are documented in Chapter 4.6.

To add an Ethernet, SLIP or PPP network interface, you must call KwikNet procedure *kn_ifadd()*, identifying the interface device driver. The device driver definition provides the same set of parameters used to define prebuilt networks using the KwikNet Configuration Builder. Once added, the interface remains in place until KwikNet is shut down, if ever. Note that a Treck network interface cannot be removed.

If a SLIP or PPP network interface requires modem support, you must call procedure *kn_ifmodem()* to attach the KwikNet Modem Driver to the interface and specify its operating characteristics.

If you require notification whenever the network interface is opened, you must call procedure *kn_ifnethook()* to install a pointer to your network startup hook function. Your function must operate as described in Appendix A.1 of the KwikNet Device Driver Technical Reference Manual.

Once a network interface has been added, you can call Treck functions to customize the interface option settings if necessary to meet your particular operating requirements.

The network interface is opened with a call to *kn_ifopen()*. When you open the interface, you must provide its network parameters such as the local IP address, subnet mask and default gateway IP address. These parameters match those used to define prebuilt networks using the KwikNet Configuration Builder. When an Ethernet network is opened, KwikNet will automatically start DHCP, BOOTP or Auto IP negotiation to acquire a network IP address if the interface has been so configured. When a PPP network is opened, KwikNet will automatically negotiate the PPP connection according to the requirements specified by you when you opened the interface.

The network interface is closed with a call to *kn_ifclose()*. Once closed, the device driver and modem driver are detached from the network interface. The interface remains closed until you next request that it be opened.

Network Descriptor

When a network interface is added with a call to *kn_ifadd()*, KwikNet allocates a network descriptor structure which it uses to define the network. A pointer to the structure is returned to the caller as a handle for subsequent references to the interface. The handle is called a network descriptor pointer and is declared to be of type *KN_NETDP*.

```
/* KwikNet Network Descriptor Structure */

struct knx_netd {
    char          xnd_tag[KN_TAGSIZE]; /* Network tag string */
    unsigned short xnd_ifnum;          /* Interface number */
    unsigned short xnd_iftype;         /* Interface type */
    ttUserInterface xnd_interface;     /* Interface handle */
    unsigned long  xnd_userinfo;       /* User defined parameter */
    struct knx_dvcprep *xnd_prepp;     /* Device preparation pointer */
};
```

If *netdp* is a network descriptor pointer, then *netdp* can be used to access the members of the network descriptor structure, as in *netdp->xnd_iftype*.

Field *xnd_tag* is the network tag assigned by you to identify your network interface.

Field *xnd_ifnum* is the interface number, a decimal number from 0 to *n-1* where *n* is the maximum number of network interfaces which your KwikNet configuration supports.

Field *xnd_iftype* is the interface type:

<i>KNIF_ETHER</i>	for Ethernet (Ethernet-II framing only),
<i>KNIF_ET8023</i>	for Ethernet (802.3 and possibly Ethernet-II framing),
<i>KNIF_SLIP</i>	for SLIP,
<i>KNIF_PPPC</i>	for a PPP client and
<i>KNIF_PPPS</i>	for a PPP server.

Field *xnd_interface* is the Treck interface handle assigned by Treck to identify the network interface. This interface handle must be used to identify the network if you call Treck functions to manipulate the network in some special way.

Field *xnd_userinfo* is reserved for the private use of your application. Once you have added a network interface, you can install an application specific parameter into this field. Since the network descriptor pointer is passed to your network callback procedure, you can use this parameter to provide access to application dependent information specific to a particular network interface.

Field *xnd_prepp* is the device preparation structure used for intercommunication between the KwikNet network driver and the associated device driver. You will require this parameter if your application must make an *ioctl* call directly to the device driver. Rarely will you have any such need.

Warning

You may modify field *xnd_userinfo* in the network descriptor structure. Do not modify any other fields.

Network Parameter Structure

When a network interface is opened with a call to *kn_ifopen()*, KwikNet configures the interface according to your definition of the operating characteristics required for that network. The parameters are presented in a network parameter structure. A separate structure is defined for each type of network interface in KwikNet header file *KN_API.H*.

Note

All of the parameters in the network parameter structure can be overridden by your network hook function which will be called by KwikNet as it prepares to open the network.

Network Parameter Structure (Ethernet)

```
struct knx_np_ether {  
    struct in_addr  xp_ether_ipaddr;      /* Network IP address      */  
    struct in_addr  xp_ether_snmask;      /* Network subnet mask     */  
    struct in_addr  xp_ether_defgate;     /* Network default gateway */  
    short          xp_ether_ipattr;      /* IP address attributes   */  
    short          xp_ether_rsv1;        /* Reserved for alignment  */  
};
```

For Ethernet networks, the IP address, subnet mask and default gateway, if not 0, will be used as the settings for the local network interface.

Field *xp_ether_ipattr* must be one of the following IP attributes flags:

0	Use the configured IP address, subnet mask and gateway
<i>KN_IPATTR_DHCP</i>	Use DHCP for IP address acquisition
<i>KN_IPATTR_REBOOT</i>	Use DHCP (reboot) for IP address acquisition
<i>KN_IPATTR_BOOTP</i>	Use BOOTP for IP address acquisition
<i>KN_IPATTR_AUTOIP</i>	Use Auto IP for IP address acquisition

If DHCP is used to dynamically acquire an IP address, the IP address is simply a preference which, if present, will be requested from the DHCP server. If DHCP (reboot) is used, the IP address must be provided for presentation in the request to the DHCP server. If BOOTP or Auto IP is used, the configured IP address is ignored.

In all cases, if an IP address is dynamically acquired, the configured subnet mask is ignored and the subnet mask derived by the DHCP, BOOTP or Auto IP process is used.

If DHCP, DHCP (reboot) or BOOTP is used, the default gateway, if any, provided by the DHCP or BOOTP server is installed as the default gateway. If the DHCP or BOOTP server does not provide a default gateway, the configured default gateway, if provided, is installed as the default gateway. In any case, the default gateway is only installed if a valid local IP address is available. Note that the first default gateway to be installed becomes the system's default gateway. When a subsequent network interface is opened, its default gateway will not override the previously installed gateway.

Network Parameter Structure (SLIP)

```
struct knx_np_slip {
    struct in_addr  xp_slip_ipaddr;          /* Network IP address      */
    struct in_addr  xp_slip_ipremote;        /* Remote peer IP address  */
    struct in_addr  xp_slip_unused;          /* Reserved field          */
    short           xp_slip_ipattr;          /* IP address attributes   */
    short           xp_slip_rsv1;            /* Reserved for alignment  */
};
```

For SLIP networks, a valid local IP address must be defined.

A remote peer IP address must be provided for routing purposes. If the remote peer IP address is specified as 0.0.0.0 when the interface is opened, then you must provide a network hook function to dynamically override the setting with a valid IP address.

If mask bit `KN_IPATTR_GATEWAY` in field `xp_slip_ipattr` is set, then the remote peer IP address will be installed as the default gateway once the SLIP connection is established. Note that the first default gateway to be installed becomes the system's default gateway. When a subsequent network interface is opened, its default gateway will not override the previously installed gateway.

Network Parameter Structure (PPP)

```
struct knx_np_ppp {
    struct in_addr  xp_ppp_ipaddr;           /* Network IP address      */
    struct in_addr  xp_ppp_ipremote;         /* Remote peer IP address  */
    struct in_addr  xp_ppp_unused;          /* Reserved field          */
    short           xp_ppp_ipattr;           /* IP address attributes   */
    short           xp_ppp_rsv1;             /* Reserved for alignment  */
    unsigned long   xp_ppp_options;         /* Negotiation options     */
    struct in_addr  xp_ppp_dns1;            /* Primary DNS IP address  */
    struct in_addr  xp_ppp_dns2;            /* Secondary DNS IP address */
};
```

For PPP networks, the network parameter definitions are somewhat complex. All of these parameters have been described in detail in Chapter 2.6. You are advised to review that chapter. Alternatively, run the KwikNet Configuration Builder, select the Networks property page, click on the **Add** button, pick PPP Client from the Network driver pull down list and click on the PPP Options... button. Press F1 for help and view the description of each option in the dialog box. These options correspond with the fields in structure *knx_np_ppp*.

If mask bit *KN_IPATTR_GATEWAY* in field *xp_ppp_ipattr* is set, then the remote peer IP address will be installed as the default gateway once the PPP connection is established. Note that the first default gateway to be installed becomes the system's default gateway. When a subsequent network interface is opened, its default gateway will not override the previously installed gateway.

Other boolean options are specified in field *xp_ppp_options* by ORing the value 0 with the applicable PPP option bit masks from KwikNet header file *KN_API.H*.

LCP negotiation:

<i>KN_PPP_HDRCOMP</i>	Use header field compression
<i>KN_PPP_MAGIC</i>	Use magic number negotiation

Authentication:

<i>KN_PPP_REAP</i>	Require EAP
<i>KN_PPP_RMSCHAP1</i>	Require MS-CHAP v1
<i>KN_PPP_RCHAP</i>	Require CHAP
<i>KN_PPP_RPAP</i>	Require PAP
<i>KN_PPP_OEAP</i>	Offer EAP
<i>KN_PPP_OMSCHAP1</i>	Offer MS-CHAP v1
<i>KN_PPP_OCHAP</i>	Offer CHAP
<i>KN_PPP_OPAP</i>	Offer PAP

IPCP Negotiation:

<i>KN_PPP_IPCOMP</i>	Use IP header compression
<i>KN_PPP_REQDNS</i>	Request DNS server information from peer

Network Attributes

Each network parameter structure includes a field which is used to specify the network interface attributes. The structure name, field name and allowable attributes are dependent upon the interface type.

Interface	Structure	Field
Ethernet	<i>knx_np_ether</i>	<i>xp_ether_ipattr</i>
SLIP	<i>knx_np_slip</i>	<i>xp_slip_ipattr</i>
PPP	<i>knx_np_ppp</i>	<i>xp_ppp_ipattr</i>

The network attributes are specified by ORing the value *o* with one or more of the following attribute bit masks from KwikNet header file *KN_COMN.H*.

<i>KN_IPATTR_DHCP</i>	Ethernet: Use DHCP for IP address acquisition
<i>KN_IPATTR_REBOOT</i>	Use DHCP (reboot) for IP address acquisition
<i>KN_IPATTR_BOOTP</i>	Use BOOTP for IP address acquisition
<i>KN_IPATTR_AUTOIP</i>	Use Auto IP for IP address acquisition
 <i>KN_IPATTR_GATEWAY</i>	SLIP and PPP: Set default gateway to remote peer's IP address
 <i>KN_IPATTR_IPFW</i>	Ethernet, SLIP and PPP: Allow IP forwarding (unicast)
<i>KN_IPATTR_IPFWBCAST</i>	Allow IP forwarding (directed broadcast)
 <i>KN_IPATTR_IPUSEALL</i>	Ethernet and PPP: IPv4 and IPv6 (both needed)
<i>KN_IPATTR_IPUSEANY</i>	IPv4 and/or IPv6 needed

Network States

A network interface is always in one of three states. The interface is **down** when first added to the KwikNet list of network interfaces. The network goes **in transit** as soon as the network is opened. The network stays in transit until a valid IP address has been assigned to the network interface at which point the network is declared **up**. The network remains up until you close it at which time it once again goes in transit. The network stays in transit until the device driver (and modem driver) have been closed and detached from the interface at which point the interface once again is considered down.

If any network interface fails to successfully open and go up, the interface will be closed and the network will revert to the down state. Note that an Ethernet network interface may stay in transit indefinitely if its DHCP, BOOT or Auto IP negotiation fails to acquire an IP address.

If an Ethernet network loses its lease on an IP address granted by a DHCP server, the interface will go in transit as it once again attempts to acquire the same IP address or, failing that, acquire a new IP address. Only if an IP address is eventually acquired will the interface once again be declared up.

When a SLIP or PPP network is opened, it will revert to the down state if it cannot successfully initialize its device driver. It will also reenter the down state if a modem is being used and the modem cannot establish a connection with the remote system. A PPP interface will go down if it cannot successfully negotiate a PPP connection with its peer.

If a SLIP or PPP network interface with a modem connection is up, it will go in transit if the modem loses its connection. The network interface will be automatically closed and the network will eventually be declared down.

If a PPP connection is closed by the network's PPP peer, the interface will go in transit. The PPP connection will be torn down, the interface will be closed and the network will eventually be declared down.

Monitoring Network Events

Your application can use KwikNet procedure *kn_ifstate()* to determine the current operating state of a network interface. Note that the state information does not indicate whether an interface that is in transit is going up or down. To gain this information, you must provide a network event notification function to monitor network state transitions as they occur. The prototype for this function is as follows:

```
void kn_netevent(KN_NETDP netdp, int eventid);
```

This function receives the network descriptor pointer identifying the network interface on which a significant event has occurred and an event identifier. Events include network interface state transitions as well as DHCP and Auto IP success and failure notifications. The event identifiers are specified in the description of this procedure in Chapter 4.6. Note that although this function is one that you must write, it is documented as though it were a KwikNet procedure.

Starting and Stopping KwikNet

Most applications start KwikNet with a call to *kn_enter()* and allow KwikNet to run forever. Some use *kn_exit()* to stop KwikNet in preparation for a termination of the entire application. Others find it necessary to stop and then restart KwikNet to recover from serious network faults. The ability to start and stop KwikNet on demand is also useful when testing your network application.

KwikNet is started with a call to *kn_enter()*. Procedure *kn_state()* can then be used to detect when KwikNet is fully operational.

KwikNet can only be stopped if you enabled KwikNet to shut down. This feature is enabled by a configuration parameter in the KwikNet Network Parameter File used in the construction of the KwikNet Library as described in Chapter 2.3. When you define your KwikNet configuration, simply check the option box labeled "KwikNet can be shut down" on the General property page.

Before KwikNet can be stopped, your application must cease using all KwikNet services. You must ensure that all KwikNet resources such as UDP channels and TCP or UDP sockets have been relinquished. Do not forget that all dedicated KwikNet clients and servers (such as those for FTP, HTTP, TELNET, TFTP, SMTP and SNMP) must have been stopped in an orderly fashion. The private KwikNet DHCP and DNS clients will be stopped by KwikNet.

KwikNet is stopped in two steps. The first step is to wait long enough for all unfinished network transactions to complete. KwikNet procedure *kn_godown()* provides this service. Your application can call *kn_godown()* to start the shutdown process and wait, up to some maximum interval, for the process to complete.

The second step is to force KwikNet to shut down all networks and their device drivers and release all memory and operating system resources. This process is initiated with a call to *kn_exit()*. Procedure *kn_exit()* does not return to the caller until KwikNet has fully stopped. Then, and only then, can KwikNet be restarted.

If your application does not call *kn_godown()* to shut down KwikNet before calling *kn_exit()*, KwikNet will automatically attempt a shutdown, waiting up to two minutes for the process to complete before finally stopping. In this case, if the shutdown fails, KwikNet will initiate a fatal exit.

In a multitasking system, procedures *kn_godown()* and *kn_exit()* can only be called from an application task executing at lower priority than the KwikNet Task. In a single threaded system, the procedures must be called from your App-Task.

Warning!

You must not shut down KwikNet unless KwikNet's memory is allocated from a static array or from a fixed region of memory acquired when KwikNet is started and released when KwikNet is stopped.

4.6 KwikNet Library Services

The KwikNet Library provides a set of network services from which the real-time system designer can choose. Many of the services are optional and, if not used or configured into your KwikNet Library, will not even be present in your final KwikNet system.

The following list summarizes the KwikNet low level service procedures which are accessible to the user. These procedures are all present in the KwikNet Library. They are grouped functionally for easy reference.

<i>kn_enter</i>	Launch the KwikNet TCP/IP Stack
<i>kn_exit</i>	Terminate the KwikNet TCP/IP Stack
<i>kn_godown</i>	Initiate a shutdown of the KwikNet TCP/IP Stack
<i>kn_state</i>	Sense the operating state of the KwikNet TCP/IP Stack
<i>kn_panic</i>	Generate a KwikNet fatal error
<i>kn_yield</i>	Yield to the KwikNet Task (single threaded use only)
<i>kn_addserver</i>	Install (add) a server function (single threaded use only)
<i>kn_fmt</i>	Format a text string
<i>kn_dprintf</i>	Format and log a text message
<i>kn_logbuffree</i>	Free a KwikNet log buffer
<i>kn_netevent</i>	Notify application of a significant network event (this function must be provided by your application)
<i>kn_netstats</i>	Log KwikNet network statistics
<i>kn_ifadd</i>	Add a network interface
<i>kn_ifopen</i>	Open a network interface
<i>kn_ifclose</i>	Close a network interface
<i>kn_ifmodem</i>	Attach the modem driver to a network interface
<i>kn_ifnethook</i>	Register a network hook procedure for a network interface
<i>kn_ifaddress</i>	Get the IP address of a local network interface
<i>kn_iffind</i>	Find a network interface with a specific network tag
<i>kn_ifnext</i>	Find next available network interface
<i>kn_ifinfo</i>	Fetch information about a network interface
<i>kn_ifstate</i>	Query the state of a network interface
<i>kn_inet_addr</i>	Convert a dotted decimal IP address to numeric form
<i>kn_inet_ntoa</i>	Convert a numeric IP address to dotted decimal string form
<i>kn_udpopen</i>	Open a UDP channel to send/receive UDP datagrams on a network
<i>kn_udpclose</i>	Close a UDP channel
<i>kn_udpbind</i>	Bind a local IP address and port to a UDP channel
<i>kn_udpsend</i>	Send a UDP datagram on a network
<i>kn_udpfree</i>	Free a received UDP message packet
<i>kn_cksum</i>	Compute an IP checksum

...more

The following BSD-like services are also available in the KwikNet IP Library.

<i>gethostbyname</i>	Get the IP address of a host with a specific domain name
<i>netlong</i> = <i>htonl(hostlong)</i>	Convert <i>long</i> from host to network endian form
<i>netshort</i> = <i>htons(hostshort)</i>	Convert <i>short</i> from host to network endian form
<i>hostlong</i> = <i>ntohl(netlong)</i>	Convert <i>long</i> from network to host endian form
<i>hostshort</i> = <i>ntohs(netshort)</i>	Convert <i>short</i> from network to host endian form

KwikNet Procedure Descriptions

A description of all KwikNet low level service procedures is provided in this chapter. The descriptions are ordered alphabetically for easy reference. All of the KwikNet procedures are described using the C programming language.

Italics are used to distinguish programming examples. Procedure names and variable names which appear in narrative text are also displayed in italics. Occasionally a lower case procedure name or variable name may appear capitalized if it occurs as the first word in a sentence.

Vertical ellipses are used in program examples to indicate that a portion of the program code is missing. Most frequently this will occur in examples where fragments of application dependent code are missing.

```
:  
: /* Continue processing */  
:
```

Capitals are used for all defined KwikNet filenames, constants and error codes. All KwikNet procedure, structure and constant names can be readily identified according to the nomenclature introduced in Chapter 1.3.

A consistent style has been adopted for the description of the KwikNet procedures presented in Chapters 4.6 and 5.4. The procedure name is presented at the extreme top right and left as in a dictionary. This method of presentation has been chosen to make it easy to find procedures since they are ordered alphabetically.

Purpose A one-line statement of purpose is always provided.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

This block is used to indicate which application procedures can call the KwikNet procedure. A filled in box indicates that the procedure is allowed to call the KwikNet procedure. In the above example, only tasks would be allowed to call the procedure.

For AMX users, this block is used to indicate which of your AMX application procedures can call the KwikNet procedure. You are reminded that the term ISP refers to the Interrupt Handler of a conforming ISP.

...more

KwikNet Procedure Descriptions (continued)

Used by

■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

For other multitasking systems, a task is any application task executing at a priority below that of the KwikNet Task. A Timer Procedure is a function executed by a task of higher priority than the KwikNet Task. An ISP is a KwikNet device driver interrupt handler called from an RTOS compatible interrupt service routine. The other procedures do not exist.

For a single threaded system, your App-Task (see glossary in Appendix A) is the only task. An ISP is a KwikNet device driver interrupt handler called from an interrupt service routine. The other procedures do not exist.

Setup

The prototype of the KwikNet procedure is shown.
The KwikNet header file in which the prototype is located is identified.
Include KwikNet header file *KN_LIB.H* or *KN SOCK.H* for compilation.

File *KN_LIB.H* is the KwikNet Library Header File used to compile the KwikNet Library which your application uses. This file is created for you by the KwikNet Configuration Builder when you create your KwikNet Library. File *KN_LIB.H* automatically includes the correct subset of the KwikNet and Treck header files for a particular target processor.

File *KN SOCK.H* is the KwikNet include file which you must include if your application uses the KwikNet TCP/IP sockets API. This file can be found in the Treck installation directory *TRECK\INCLUDE*. File *KN SOCK.H* automatically includes file *KN_LIB.H* if it has not already been included.

Description Defines all input parameters to the procedure and expands upon the purpose or method if required.

Returns The outputs, if any, produced by the procedure are always defined. Most KwikNet procedures return an integer error status. Additional TCP/IP socket error information is also available via KwikNet procedure *kn_errno()*.

Restrictions If any restrictions on the use of the procedure exist, they are described.

Note Special notes, suggestions or warnings are offered where necessary. The following paragraph is an example of such a note.

All KwikNet procedures assume that an integer or unsigned integer is a 16 or 32-bit value dependent only upon the basic register width of the target processor.

Example In many cases, a simple example is provided. The examples are kept simple and are intended only to illustrate the correct calling sequence.

See Also A cross reference to other related KwikNet procedures is always provided if applicable.

hton-
ntoh-

hton-
ntoh-

Purpose **Convert Between Host and Network Endian Forms**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup The macro definitions are in file *KN_LIB.H*.
C dependent, in-line assembly language expansions are in file *KNZZZCC.H*.
#include "KN_LIB.H"

Convert 32-bit values

netlong = *htonl(hostlong)*
hostlong = *ntohl(netlong)*

Convert 16-bit values

netshort = *htons(hostshort)*
hostshort = *ntohs(netshort)*

Description *Hostlong* is any 32-bit value in host endian form.
Netlong is any 32-bit value in net endian form.
Hostshort is any 16-bit value in host endian form.
Netshort is any 16-bit value in net endian form.

If the KwikNet Library has been configured for big endian operation, these macros do nothing since the input values require no conversion.

If the KwikNet Library has been configured for little endian operation, these macros may expand to a function call, an in-line function expansion or a series of C statements, depending upon which C compiler is being used.

The goal is always to ensure the fastest possible execution of these frequently encountered macros. When possible, these macros have been implemented using in-line assembly language statements generated by the C compiler. In some cases, the macros generate calls to assembly language functions of a form supported by the C compiler. As a last resort, the macros expand to a series of in-line C statements.

Returns The input value converted to opposite endian form.

Restriction These macros can introduce side effects. Therefore, the macro parameters must not use expressions which include operators such as *--* or *++* since they always produce side effects. You must also avoid using expressions which include function calls to fetch parameters if the functions can introduce side effects.

Example See examples in the descriptions of *kn_cksum()*, *kn_dprintf()* and *kn_fmt()*.

Purpose **Install (Add) a Server Function****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_API.H*.
 #include "KN_LIB.H"
 *void kn_addserver(struct knx_svbblock *servercbp,*
 unsigned long period,
 *int (*serverfnp)(void *), void *param);*

Description *Servercbp* is a pointer to a KwikNet server control block which KwikNet can use to control the periodic execution of this server function. Each server function requires its own unique server control block.

You can create a server control block by declaring it as a unique structure variable or by including it as a *knx_svbblock* structure within some other structure variable. The variable must reside outside all functions in the source module. The call to *kn_addserver()* must provide a pointer to the instance of the structure.

Parameter *period* specifies the interval at which the server function is to be executed by the KwikNet Task. The interval, measured in milliseconds, is converted to a non-zero, integral multiple of KwikNet ticks. The server function is therefore executed at the resulting period, measured in equivalent KwikNet ticks.

If parameter *period* is 0, the server function will be executed by the KwikNet Task whenever a KwikNet clock tick or significant event is serviced. Hence, at a minimum, the server function will execute at the KwikNet clock frequency. However, it will also execute if, at the time your App-Task yields to the KwikNet Task, other stack related services are pending.

Parameter *serverfnp* is a pointer to the server function to be executed by the KwikNet Task. The function is called with a single parameter, a copy of parameter *param* presented in the call to *kn_addserver()*.

The server function must return the value 0 in order to remain on the active server list, ready to be executed at its specified period. If the server function returns a non-zero value, the server will be removed from the KwikNet server list.

Returns Nothing

Restriction This function must only be used in a single threaded system. It can be called while executing in either the user or KwikNet domain.

See Also *kn_yield()*

Purpose **Compute an IP Checksum****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
unsigned short kn_cksum(void *p, unsigned int n);
```

Description *P* is a pointer to a 16-bit aligned region of memory containing an array of 16-bit *unsigned short* integers to be checksummed.*N* is the number of 16-bit *unsigned short* integers in the memory array referenced by pointer *p*.**Returns** The 16-bit, unsigned IP checksum of the *n* unsigned short integers in the memory array referenced by pointer *p*. Note that the computed checksum is not complemented before being returned to the caller.

Each 16-bit, unsigned short integer is added to the checksum using ones complement arithmetic. Any overflow (carry) from the 16-bit checksum is repetitively added to the checksum until no further overflow occurs.

The algorithms used by KwikNet to implement this procedure are both processor and compiler dependent. The procedure has been coded for fastest possible execution. If the C compiler supports the use of assembly language within C, the procedure is coded in C using assembly language statements of the form supported by the C compiler. Otherwise, the function is coded reasonably efficiently using only C language statements.

Note The checksum algorithm is impervious to the processor's endianness. Hence the 16-bit IP checksum can be stored into and read from the IP packet header without conversion between net and host endianness as illustrated in the example.**Example**

```
#include "kn_lib.h"

unsigned short cksum; /* Computed checksum */
unsigned short pktsum; /* Packet checksum */
unsigned short *p; /* IP header pointer */

pktsum = *(p + 5); /* Save IP header checksum */
*(p + 5) = 0; /* Checksum = 0 in packet */

/* Compute IP header checksum */
/* Length = IHL*2 shorts */
cksum = ~kn_cksum(p, (ntohs(*p) & 0x0F) << 1);

if (cksum != pktsum) {
    kn_dprintf(0, "Received packet has checksum error.\n");
    kn_dprintf(0, "Received %4X; expected %4X.\n",
               ntohs(pktsum), ntohs(cksum));
}
```

Purpose **Format and Log a Text Message****Used by** ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_dprintf(int attrib, const char *fmt, ...);
```

Description *Attrib* is a parameter which defines the message attributes. Applications should use the value of 0 for *attrib*. Valid attributes are described in Chapter 1.6.

Fmt is a pointer to a format specification string similar to that expected by the C library procedure *printf()*. Allowable format specifications are summarized in the description of procedure *kn_fmt()*.

The format string is followed by zero or more parameters of types specified by the format string.

Returns Nothing**Example**

```
#include "kn_lib.h"

char      *bufp;                /* Input buffer pointer      */
struct in_addr ipaddr;          /* IPv4 address (numeric)    */
char      ipstring[40];         /* IPv4 addr (dotted decimal)*/

bufp = "192. 168. 5";           /* An unusual input string   */

if (kn_inet_addr(bufp, &ipaddr) != 1)
    kn_dprintf(0, "Conversion of '%s' to 0xC0A80500" \
                "(192.168.5.0) failed.\n", bufp);

else if (kn_inet_ntoa(&ipaddr, ipstring) !=
        strlen("192.168.5.0"))
    kn_dprintf(0, "Conversion to '192.168.5.0' failed.\n");

else {
    kn_dprintf(0, "Converted '%s' to 0x%08lx to '%s'.\n",
                bufp, ntohl(ipaddr.s_addr), ipstring);

    /* The previous message should read:
    /* "Converted '192. 168. 5' to 0xc0a80500
    /*                               to '192.168.5.0'."
    */
}
```

See Also *kn_fmt()*, *kn_netstats()*

kn_enter **kn_exit**

kn_enter **kn_exit**

Purpose **Launch or Terminate the KwikNet TCP/IP Stack**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_enter(void);
void kn_exit(void);
```

Description Procedure *kn_enter()* must be called to launch (start) the KwikNet TCP/IP Stack.

Procedure *kn_exit()* must be called to terminate (stop) the KwikNet TCP/IP Stack.

In a multitasking system, procedure *kn_enter()* must be called before any task can use KwikNet services. Procedure *kn_exit()* must not be called until all tasks, including KwikNet client and server tasks, have stopped using KwikNet services.

In a single threaded system, procedure *kn_enter()* must be called by the App-Task. Procedure *kn_exit()* must not be called until the App-Task and all active clients and servers, including KwikNet clients and servers, have stopped using KwikNet services.

Returns Nothing

Restriction A fatal KwikNet error will occur if you call procedure *kn_exit()* and your KwikNet Library does not have KwikNet shut down enabled. Edit your KwikNet Network Parameter File and check the option box labeled "KwikNet can be shut down" on the General property page. Then rebuild your KwikNet Library.

Restriction You must not shut down KwikNet unless KwikNet's memory is allocated from a static array or from a fixed region of memory acquired when KwikNet is started and released when KwikNet is stopped. If you do so and then try to restart KwikNet by calling *kn_enter()*, KwikNet will initiate a fatal exit.

AMX Note Procedure *kn_enter()* can be treated as if it is an AMX Restart Procedure. Alternatively, it can be called from a Restart Procedure or from an application task.

Procedure *kn_exit()* can be treated as if it is an AMX Exit Procedure. Alternatively, it can be called from an Exit Procedure or from an application task which is executing on behalf of an AMX Exit Procedure.

See Also *kn_godown()*, *kn_panic()*

Purpose **Format a Text String****Used by** ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_fmt(char *bufp, const char *fmt, ...);
```

Description *Bufp* is a pointer to storage for the formatted string.

Fmtp is a pointer to a format specification string similar to that expected by the C library procedure *printf()*. Allowable format specifications are summarized on the next page.

The format string is followed by zero or more parameters of types specified by the format string.

Returns The formatted string is stored at **bufp* and the length of that string is returned. The length is a positive value. The string is terminated with a '\0' character.**Example** Other examples are provided in the descriptions of *kn_cksum()* and *kn_dprintf()*.

```
#include "kn_lib.h"

struct in_addr ipaddr;          /* IPv4 address (numeric)    */
char          buf[80];          /* String buffer            */

/* IP address = 192.168.5.21 */
ipaddr.s_addr = htonl(0xC0A80516);

if (kn_fmt(buf, "IP address 0x%08lX is '%03a'.\n",
           ntohl(ipaddr.s_addr), ipaddr.s_addr) <= 0)
    kn_dprintf(0, "Cannot convert IP address to string.\n");
else {
    kn_dprintf(0, "%s", buf);

    /* The previous message should read:
    /* "IP address 0xC0A80516 is '192.168.005.016'."
    */
}
```

See Also *kn_dprintf()*

...more

Formats ...continued

Allowable format specification strings must be of the form "...%-0##z?..."

where:

% = format specification leadin character; use %% for % in output string

- = left justify in field

0 = zero fill

= field width as decimal value

z = l if variable is *long* (? is one of d, u, x, X)

z = h if variable is *short int* (? is one of d, u, x, X)

z = h for an IP address in hex format (? is one of a, A)

? = one of following field variable specifications

c = character value

d = decimal integer value

u = unsigned decimal integer value

x = unsigned hexadecimal integer value (use a .. f)

X = unsigned hexadecimal integer value (use A .. F)

f = fill with next character to field width

p = pointer value

s = string value

a = internet IPv4 address as "1.26.3.127"

"%03a" yields "001.026.003.127"

ha = internet IPv4 address as "01.1a.03.ff"

"%hA" yields "01.1A.03.FF"

"%0##a" yields "001.002.003.004"

for any non-zero value "0##".

"%0##ha" yields "0a.0b.0c.0d"

for any non-zero value "0##".

S = copy characters from format string into the output field

until ' ' is encountered

"%20Sabc`" right justifies "abc" in a 20 character field.

"%-20Sabc`" left justifies "abc" in a 20 character field.

Numeric fields that overflow the field width are formatted as xxxxxx to the full width of the field.

Purpose	Initiate a Shutdown of the KwikNet TCP/IP Stack
Used by	■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure
Setup	Prototype is in file <i>KN_API.H</i> . <pre>#include "KN_LIB.H" int kn_godown(int timeout);</pre>
Description	<p><i>Timeout</i> is the maximum interval, measured in seconds, which the caller is willing to wait for the KwikNet shutdown to complete. If <i>timeout</i> is 0, the caller will wait forever or until an error condition is detected.</p> <p>In a multitasking system, procedure <i>kn_godown()</i> must not be called until all tasks, including KwikNet client and server tasks, have stopped using KwikNet services.</p> <p>In a single threaded system, procedure <i>kn_godown()</i> must be called by the App-Task. Procedure <i>kn_godown()</i> must not be called until the App-Task and all active clients and servers, including KwikNet clients and servers, have stopped using KwikNet services.</p>
Returns	If successful, a value of 0 is returned. On failure, the error status 1 is returned. If a shutdown has already been initiated, a value of -1 is returned.
Restriction	Procedure <i>kn_godown()</i> must be called to shut down the KwikNet TCP/IP Stack prior to calling <i>kn_exit()</i> to terminate operation of the stack. If this restriction is not met, KwikNet will automatically call this function with a 2 minute (120 second) timeout interval when <i>kn_exit()</i> is called.
Restriction	This procedure name will be unresolved if you reference this procedure and your KwikNet Library does not have KwikNet shut down enabled. Edit your KwikNet Network Parameter File and check the option box labeled "KwikNet can be shut down" on the General property page. Then rebuild your KwikNet Library.
AMX Note	Procedure <i>kn_godown()</i> can be called from an Exit Procedure or from an application task which is executing on behalf of an AMX Exit Procedure.
See Also	<i>kn_enter()</i> , <i>kn_exit()</i> , <i>kn_panic()</i>

Purpose **Add a Network Interface****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
KN_NETDP kn_ifadd(KN_FNP_NETPREP netprepp,
                  const struct knx_dvcdef *ddefp);
```

Description *netprepp* is a pointer to a network preparation function which will be used by KwikNet to prepare a network interface for use. Set parameter *netprepp* to one of the following values to identify the type of network interface that you wish to add.

<i>KN_ND_ETHERNET</i>	Ethernet network interface (Ethernet-II framing)
<i>KN_ND_ET8023</i>	Ethernet network interface (802.3 framing and possibly Ethernet-II framing)
<i>KN_ND_SLIP</i>	SLIP network interface
<i>KN_ND_PPPC</i>	PPP Client network interface
<i>KN_ND_PPPS</i>	PPP Server network interface

ddefp is a pointer to a device definition structure which identifies the device driver to be attached to the network interface. This structure is described in Chapter 2.2 of the KwikNet Device Driver Technical Reference Manual. This structure defines the network tag and the device driver initialization parameters.

Returns If the network interface is successfully added, a network descriptor pointer is returned.

If the network interface cannot be added for any reason, a *NULL* network descriptor pointer is returned.

Note Your network event function (see *kn_netevent()*) will be called with an indication that this new network interface has been added.**See Also** *kn_ifclose()*, *kn_ifopen()*

Purpose **Get the IP Address for a Network Interface**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_ifaddress(KN_NETDP netdp, struct in_addr *inaddrp);
```

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the network interface of interest.

Inaddrp is a pointer to a structure into which the network's IPv4 address in net endian form will be stored. The BSD structure *in_addr* is defined in Treck header file *TRSOCKET.H* as follows:

```
struct in_addr {
    u_long s_addr;           /* IP address (net endian) */
};
```

Returns If successful, a value of 0 is returned.

On failure, one of the following error status codes is returned:

<i>KN_ERPARAM</i>	Parameter <i>netdp</i> does not refer to a valid network or parameter <i>inaddrp</i> is <i>NULL</i> .
<i>KN_ERNETWORK</i>	The network interface is not up and hence does not yet have a valid IP address.

On failure, the storage at **inaddrp* is unaltered.

See Also *kn_ifinfo()*, *kn_netstats()*

Purpose **Close a Network Interface****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_ifclose(KN_NETDP netdp);
```

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the network interface of interest.**Returns** If the network interface is successfully closed or was already closed, a value of 0 is returned.

If the network interface is in transit but not yet closed, a warning status of *KN_WRPROGRESS* is returned.

On failure, one of the following error status codes is returned:

<i>KN_ERPARAM</i>	Parameter <i>netdp</i> does not refer to a valid network.
<i>KN_ERNETWORK</i>	An unexpected network error occurred.
	The network is down but may not be reusable.

Note You can close a prebuilt network interface and then open it with an alternate set of network configuration parameters. Of course, you can also reopen the network interface with the configuration unaltered.**Note** Your network event function (see *kn_netevent()*) will be called allowing you to monitor each network state transition.**See Also** *kn_ifadd()*, *kn_ifopen()*

kn_iffind

kn_iffind

Purpose Find a Network Interface with a Specific Network Tag

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
KN_NETDP kn_iffind(const char *nettag);
```

Description *Nettag* is a pointer to a string which specifies the 1 to 7 character network tag of the network interface of interest.

Returns If successful, the network descriptor pointer for the network interface is returned. This KwikNet handle can be used to identify the network interface in subsequent calls to KwikNet procedures.

If a network interface with the specified tag does not exist, a *NULL* network descriptor pointer is returned.

Purpose **Fetch Information About a Network Interface****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_ifinfo(KN_NETDP netdp, struct knx_ifinfo *infop);
```

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the network interface of interest.

Infop is a pointer to a structure into which a summary of the network's current operating characteristics will be stored. Structure *knx_ifinfo* is defined in file *KN_API.H* as follows:

```
struct knx_ifinfo {
    char xif_tag[KN_TAGSIZE]; /* Network tag string */
    int xif_num; /* Interface number */
    int xif_type; /* Interface type */
    int xif_state; /* Interface state */
    /* (see KN_NIFS_xxxx flags) */
    #if (integers are less than 32 bits)
    int xif_rsv1; /* Reserved for alignment */
    #endif
    unsigned long xif_duration; /* Time in current state */
    /* (measured in seconds) */
    struct in_addr xif_ipaddr; /* IPv4 address */
    struct in_addr xif_snmask; /* Subnet mask */
};
```

Returns If successful, a value of 0 is returned.

On failure, one of the following error status codes is returned:

<i>KN_ERPARAM</i>	Parameter <i>netdp</i> does not refer to a valid network or parameter <i>infop</i> is <i>NULL</i> .
-------------------	---

On failure, the storage at **infop* is unaltered.

See Also *kn_ifaddress()*, *kn_netstats()*

Purpose **Attach the Modem Driver to a Network Interface****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_ifmodem(KN_NETDP netdp, const struct knx_mdmdf *mdefp);
```

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the network interface of interest.

Parameter *mdefp* is a pointer to a modem definition structure which specifies the configuration parameters for the modem which is attached to the network interface. The modem definition structure is described in Chapter 1.7 of the KwikNet Device Driver Technical Reference Manual. The structure is defined in file *KN_API.H*.

Note that the structure referenced by *mdefp* **must be declared *static*** and must remain constant once it is attached to the network interface. You can attach an alternate modem definition to the network interface as long as the interface is first closed. You cannot remove the modem definition once it is attached.

Returns If the modem driver is successfully attached to the network interface, a value of 0 is returned.

On failure, one of the following error status codes is returned:

<i>KN_ERPARAM</i>	Parameter <i>netdp</i> does not refer to a valid network or parameter <i>mdefp</i> is <i>NULL</i> .
<i>KN_ERLOGIC</i>	KwikNet is not configured to provide modem support.

Note The modem will not be configured and connected to the remote system until the network interface is opened.**See Also** *kn_ifadd()*, *kn_ifclose()*, *kn_ifopen()*

Purpose **Register a Network Hook Function for a Network Interface**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.
 #include "KN_LIB.H"
 int kn_ifnethook(KN_NETDP netdp, KN_FNP_NETHOOK nethookfn);

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the network interface of interest.

Parameter *nethookfn* is a pointer to your network hook function. This function is described in Appendix A.1 of the KwikNet Device Driver Technical Reference Manual. This function will be called the next time the network interface is opened.

To remove a previously installed network hook function, call this procedure with parameter *nethookfn* set to *(KN_FNP_NETHOOK)0L*.

Returns If successful, a value of 0 is returned.

On failure, one of the following error status codes is returned:

KN_ERPARAM Parameter *netdp* does not refer to a valid network or
 parameter *nethookfn* is *NULL*.

See Also *kn_ifadd()*

Purpose Find the Next Available Network Interface**Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
KN_NETDP kn_ifnext(KN_NETDP netdp);
```

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the currently selected network interface of interest.If *netdp* is *NULL*, the first available network interface will be selected.

KwikNet maintains a list of network interfaces that have been added to the system since KwikNet was last started. Since network interfaces can be added but not removed, this list is always maintained in the order in which interfaces are added. This procedure searches the list for the next available interface following the interface specified by *netdp* or starting at the head of the list if *netdp* is *NULL*.

Returns If successful, the network descriptor pointer for the next available network interface is returned. This KwikNet handle can be used to identify the network interface in subsequent calls to KwikNet procedures.If there is no network interface following the one identified by parameter *netdp*, a *NULL* network descriptor pointer is returned.**Example** This procedure can be called repetitively to identify all network interfaces and service each in some specific manner.

```
#include "kn_lib.h"

KN_NETDP netdp;                                /* Network descriptor */

netdp = NULL;
do {
    netdp = kn_ifnext(netdp);
    if (netdp && (kn_ifstate(netdp) == KN_NIFS_UP))
        servicenet(netdp);
} while (netdp != NULL);
```


Purpose **Open a Network Interface****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_ifopen(KN_NETDP netdp, void *paramp);
```

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the network interface of interest.

Parameter *paramp* is a pointer to a network parameter structure which specifies the configuration parameters for the network interface. These structures are described in Chapter 4.5. Use the network parameter structure which corresponds to the type of network interface that you are opening. These structures are defined in file *KN_API.H*.

<i>knx_np_ether</i>	Ethernet network interface
<i>knx_np_slip</i>	SLIP network interface
<i>knx_np_ppp</i>	PPP Client network interface
<i>knx_np_ppp</i>	PPP Server network interface

If parameter *paramp* is *NULL*, the network interface will be opened using the same parameters that were used the previous time the network was opened. If the network has never been opened, setting *paramp* to *NULL* is equivalent to opening the network with all network configuration parameters set to 0.

Returns If the network interface is successfully opened and is up with a valid IP address, a value of 0 is returned. If the interface is not yet up but is in transit going up, a warning status of *KN_WRPROGRESS* is returned.

On failure, one of the following error status codes is returned:

<i>KN_ERPARAM</i>	Parameter <i>netdp</i> does not refer to a valid network.
<i>KN_ERREJECT</i>	The request has been rejected. A network interface must be in the down state before it can be opened.
<i>KN_ERNETWORK</i>	An unexpected network or device error occurred. The network interface cannot be opened.

Note You can close a prebuilt network interface and then open it with an alternate set of network configuration parameters. Of course, you can also reopen the network interface with the configuration unaltered.**Note** Your network hook, device hook and modem hook functions will be called, if they exist, as the network, device driver and optional modem are made ready for use. Your network event function (see *kn_netevent()*) will also be called allowing you to monitor each network state transition.**See Also** *kn_ifadd()*, *kn_ifclose()*

Purpose **Query the State of a Network Interface****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_ifstate(KN_NETDP netdp);
```

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the network interface of interest.**Returns** If successful, one of the following constant values will be returned.
KN_NIFS_DOWN The network was down and unavailable for use.
KN_NIFS_TRANSIT The network was in the process of going up or down.
KN_NIFS_UP The network was up with a valid IP address.On failure, the value *-1* is returned.**Example** See the example provided with the description of procedure *kn_ifnext()*.**See Also** *kn_ifinfo()*, *kn_netstats()*

Purpose Convert a Dotted Decimal IP Address to Numeric Form**Used by** ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_inet_addr(const char *sp, struct in_addr *inaddrp);
```

Description *sp* is a pointer to a string containing an IPv4 address in dotted decimal form. The string does not have to be terminated by '\0'. Leading whitespace before the numbers in the IP address will be ignored. The separating dot must be the first character after each number.

Hence, "254. 76.2. 1abc" is an acceptable input string.

Inaddrp is a pointer to a structure into which the numeric IPv4 address in net endian form will be stored. The BSD structure *in_addr* is defined in Treck header file *TRSOCKET.H* as follows:

```
struct in_addr {
    u_long s_addr;                /* IP address (net endian) */
};
```

Returns 0 if the string contained a full IP address such as "75.4. 34.12abc".

1, 2 or 3 if the string contained a partial IP address such as "75.4.34", "75.4" or "75" respectively. In each case, the missing fields are assumed to be 0. For example "75.4" is assumed to be "75.4.0.0".

The resulting IPv4 address in numeric, net endian form is stored in the IPv4 address structure at *inaddrp->s_addr*.

Limited error checking is performed. Parsing stops at the first character which is not acceptable within an IP address or as soon as four dot separated values are found.

A value greater than 3 is returned if no numeric values are present. The value -1 is returned if any of the decimal values encountered are outside the range 0 to 255.

Note Unlike its BSD counterpart *inet_addr()*, this KwikNet procedure uses structure *in_addr* to hold the IP address.**Example** See example in the description of *kn_dprintf()*.**See Also** *kn_inet_ntoa()*

Purpose	Convert a Numeric IP Address to Dotted Decimal String Form
Used by	■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure
Setup	Prototype is in file <i>KN_API.H</i> . <pre>#include "KN_LIB.H" int kn_inet_ntoa(struct in_addr *inaddrp, char *sp);</pre>
Description	<p><i>Inaddrp</i> is a pointer to a structure containing an IPv4 address in net endian form. The BSD structure <i>in_addr</i> is defined in Treck header file <i>TRSOCKET.H</i> as follows:</p> <pre>struct in_addr { u_long s_addr; /* IP address (net endian) */ };</pre> <p><i>Sp</i> is a pointer to storage for the string showing the IPv4 address in dotted decimal form.</p>
Returns	<p>The formatted string is stored at <i>*sp</i> and the length of that string is returned. The length is a positive value. The string is terminated with a '\0' character.</p> <p>The IP address <i>0x7F000017</i> will produce the string "127.0.0.23".</p>
Note	Unlike its BSD counterpart <i>inet_ntoa()</i> , this KwikNet procedure is reentrant. It also uses structure <i>in_addr</i> to hold the IP address.
Example	See example in the description of <i>kn_dprintf()</i> .
See Also	<i>kn_inet_addr()</i>

kn_logbuffree

kn_logbuffree

Purpose **Free a KwikNet Log Buffer**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.
 `#include "KN_LIB.H"`
 `void kn_logbuffree(char *bufp);`

Description *Bufp* is a pointer to the KwikNet log buffer which is to be freed. This is a buffer allocated by KwikNet from its private pool of log buffers in response to a request to log a message.

 Your application log function must call *kn_logbuffree()* to free each log buffer that it accepts.

Returns Nothing

Purpose Notify Application of Significant Network Event (User Function)

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_netevent(KN_NETDP netdp, int eventid);
```

Description KwikNet calls this user function whenever a significant network event has been detected. Your application must provide this function.

Netdp is a network descriptor pointer, the KwikNet handle used to identify the network interface which has generated the significant event.

Parameter *eventid* identifies the significant event. This parameter will be one of the following constants defined in file *KN_API.H*.

<i>KN_EV_DVCRESET</i>	Device reset failure
<i>KN_EV_DVCFAIL</i>	Device failed
<i>KN_EV_DVCDOWN</i>	Device is down
<i>KN_EV_DVCUP</i>	Device is up
<i>KN_EV_MODEM</i>	Modem failed (lost carrier)
<i>KN_EV_NETADD</i>	Network interface added
<i>KN_EV_NETDOWN</i>	Network is closed
<i>KN_EV_NETUP</i>	Network is open and usable
<i>KN_EV_NETTRANS</i>	Network in transit (going up/down)
<i>KN_EV_NETFAIL</i>	Network state transition failed
<i>KN_EV_DHCPPOK</i>	DHCP or BOOTP acquired IP address
<i>KN_EV_DHCFAIL</i>	DHCP failed to acquire an IP address or renew lease
<i>KN_EV_AIPOK</i>	Auto IP acquired IP address
<i>KN_EV_AIPFAIL</i>	Auto IP failed to acquire IP address
<i>KN_EV_V6_START</i>	Duplicate Address Detection (DAD) started
<i>KN_EV_V6_FAIL</i>	IPv6 DAD failed
<i>KN_EV_V6_GOOD</i>	IPv6 DAD successfully completed
<i>KN_EV_V6_DUP</i>	Conflict on address already granted
<i>KN_EV_V6_DEPR</i>	Address deprecated
<i>KN_EV_V6_INVALID</i>	Address invalidated

Returns Nothing

Restriction This function is not a KwikNet procedure. It is a function that your application must provide. KwikNet will call your function if your KwikNet Library has network event notification enabled. To do so, edit your KwikNet Network Parameter File and check the option box labeled "Enable network event notification" on the General property page. Then rebuild your KwikNet Library.

See Also *kn_ifadd()*, *kn_ifclose()*, *kn_ifopen()*

Purpose **Log KwikNet Network Statistics****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_netstats(unsigned long statmask);
```

Description KwikNet will log network statistics which it has been gathering. The information is formatted into log buffers and presented a line at a time to your data logging function (see Chapter 1.6).

Statmask is a bit mask identifying the subset of network statistics maintained by KwikNet which you wish to log. *Statmask* can be created by ORing one or more of the bit mask constants to select the particular statistics to be logged. The constants are defined in file *KN_API.H*.

Statmask must be the logical OR of one or more of following masks:

<i>KN_NS_NET</i>	General network statistics for all networks
<i>KN_NS_DVC</i>	Device driver statistics for each network
<i>KN_NS_MEM</i>	Memory usage dump

Treck network information dumps are identified as follows. You must identify which Treck dumps you want by ORing one or more of the following masks into parameter *statmask*.

<i>KN_NS_ARP</i>	ARP table information
<i>KN_NS_RTE</i>	Routing table information
<i>KN_NS_TCP</i>	TCP socket summary
<i>KN_NS_UDP</i>	UDP socket summary
<i>KN_NS_IF</i>	Low level Treck interface summary

<i>KN_NS_INFOLIST</i>	All of the available Treck dumps
-----------------------	----------------------------------

To log all statistics, set *statmask* to *KN_NS_ALL*.

Restriction Network statistics will not be available unless enabled in your KwikNet Library. Edit your KwikNet Network Parameter File and selectively check the Log option boxes on the Debug property page. Check the box labeled "Monitor memory usage" to enable memory usage logs.**Returns** Nothing**See Also** *kn_dprintf()*, *kn_ifinfo()*, *kn_ifstate()*

Purpose **Generate a KwikNet Fatal Error**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
void kn_panic(const char *msgp);
```

Description *msgp* is a pointer to a message string to be logged by KwikNet before it initiates a fatal shutdown of the underlying RT/OS. If you are using the AMX kernel, your system will be suspended with interrupts disabled to prevent any further possibility of incorrect operation.

Returns Never

See Also *kn_exit()*

Purpose Sense the Operating State of the KwikNet TCP/IP Stack

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
unsigned int kn_state(unsigned int wstate, unsigned long mswait);
```

Description This function can be used to determine the current operating state of KwikNet or to wait until KwikNet reaches a particular state.

Parameter *wstate* identifies the particular KwikNet state or states of interest. If *wstate* is 0, the current KwikNet state will be returned to the caller without delay. In this case, parameter *mswait* will be ignored.

If parameter *wstate* is not 0, the caller will be blocked until the KwikNet operating state matches one or more of the specified state masks.

Wstate must be the logical OR of one or more of following masks:

<i>KN_TS_IDLE</i>	KwikNet is idle (not in use at all).
<i>KN_TS_START</i>	KwikNet is starting up.
<i>KN_TS_RUN</i>	KwikNet is fully operational.
<i>KN_TS_GODOWN</i>	KwikNet shutdown is in progress.
<i>KN_TS_EXIT</i>	KwikNet exit is in progress.
<i>KN_TS_STOPPED</i>	KwikNet has stopped. This state is fleeting. KwikNet will immediately enter the idle state.

Mswait is the number of milliseconds which the caller is prepared to wait for KwikNet to reach any of the specified states. A value of 0 indicates that the caller will wait forever.

Returns If *wstate* is 0, the current KwikNet state is returned.
The value will be the logical OR of one or more of the above state masks.

If *wstate* is not 0, the KwikNet state at the time of the match is returned.
If a timeout occurs or the caller cannot be blocked, the value 0 is returned.

Note States *KN_TS_IDLE*, *KN_TS_START*, *KN_TS_RUN* and *KN_TS_STOPPED* are mutually exclusive. State masks *KN_TS_GODOWN* and *KN_TS_EXIT* can be set in conjunction with other state masks.

To detect that KwikNet has fully stopped and is no longer in use, you must wait with state mask *wstate* = *KN_TS_IDLE*.

Restriction This function will not alter the state of KwikNet. This procedure must not be called by any function executing in the context of the KwikNet Task.

See Also *kn_ifstate()*

Purpose Bind a Local IP Address and Port to a UDP Channel

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_udpbind(unsigned long udphandle,
               struct in_addr *lhostp, int lport);
```

Description *Udphandle* is a UDP handle acquired with a previous call to *kn_udpopen()*.

Lhostp is a pointer to a structure containing the IPv4 address, in net endian form, of the local network interface which you wish to bind to the UDP channel. All UDP datagrams sent on the UDP channel will be transmitted on that network interface.

The BSD structure *in_addr* is defined in Treck header file *TRSOCKET.H* as follows:

```
struct in_addr {
    u_long s_addr;           /* IP address (net endian) */
};
```

Lport is the local port number, in host endian form, to be used for communication. Set *lport* to 0 if you will accept UDP datagrams destined to any port on the network interface to which the UDP channel is bound.

Returns If successful, a value of 0 is returned.

If a local network with the specified IPv4 address does not exist, the error status -1 is returned.

Restriction You cannot bind a UDP channel to a specific local network interface unless the UDP channel was opened with a local IP address and port of 0. A UDP channel can only be bound once. To bind to an alternate IP address or port, you must close the UDP channel, open another one and bind it as required.

See Also *kn_udpclose()*, *kn_udpopen()*

Purpose **Close a UDP Channel**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_udpclose(unsigned long udphandle);
```

Description *Udphandle* is a UDP handle acquired with a previous call to *kn_udpopen()*. This handle must not be used after the UDP channel which it represents has been closed.

Returns If successful, a value of *0* is returned.
 On failure, the error status *-1* is returned.

Example See example in the description of *kn_udpopen()*.

See Also *kn_udpopen()*

Purpose **Free a Received UDP Message Packet**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.
 `#include "KN_LIB.H"`
 `void kn_udpfree(struct knx_udpmsg *msgp);`

Description *msgp* is a pointer to the UDP message descriptor given to your application's UDP callback function upon receipt of a UDP datagram.

Returns Nothing

Example See example in the description of *kn_udpopen()*.

See Also *kn_udpopen()*

Purpose **Open a UDP Channel to Send/Receive UDP Datagrams on a Network**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
unsigned long kn_udpopen(struct in_addr *fhostp, int fport,
                        struct in_addr *lhostp, int lport,
                        int (*udprcv)(struct knx_udpmsg *msgp, void *userp),
                        void *userp);
```

Description *Fhostp* is a pointer to a structure containing the IPv4 address, in net endian form, of the foreign host with whom you wish to communicate using UDP datagrams. Set *fhostp* to *NULL* or use IP address 0.0.0.0 to identify any host.

Fport is the foreign port number, in host endian form, for the host with whom you intend to communicate. If *fport* is 0, then any UDP datagram from the foreign host will be accepted on the UDP channel.

Lhostp is a pointer to a structure containing the IPv4 address, in net endian form, of the local network interface which you wish to bind to this UDP channel. All UDP datagrams sent on the UDP channel will be transmitted on that network interface. Set *lhostp* to *NULL* or use IP address 0.0.0.0 to identify any local network interface.

Lport is the local port number, in host endian form, to be used for communication. If *lport* is 0, an arbitrary local port will be assigned for your UDP channel.

The BSD structure *in_addr* is defined in Treck header file *TRSOCKET.H* as follows:

```
struct in_addr {
    u_long s_addr;           /* IP address (net endian) */
};
```

If *lhostp* and *lport* are both 0, you can use procedure *kn_udpbind()* to bind the UDP channel to a specific network interface and port after the channel has been opened.

...more

Description ...continued

Udprcv is the name of your application callback function which will be called to process received UDP datagrams. Your function will receive parameter *msgp*, a pointer to a UDP message descriptor. The second parameter, *userp*, is a copy of the pointer variable *userp* received as a parameter in this *kn_udpopen()* procedure call.

Your UDP callback function must be coded as described in Chapter 4.1. The callback function must return *-1* if it cannot accept the message descriptor. It must return *0* if it accepts the UDP message descriptor. In this case, your application must accept responsibility for the UDP message descriptor and must eventually free the descriptor by calling procedure *kn_udpfree()* when finished processing the datagram.

Userp is any pointer variable which your *udprcv()* function might require. *NULL* is an acceptable value.

Returns If successful, a non-zero UDP handle is returned.
On failure, a UDP handle of *0L* is returned.

See Also *kn_udpbind()*, *kn_udpclose()*, *kn_udpfree()*
...more

...continued

Example

```
#include "kn_lib.h"

CJ_ID      udptaskid;      /* UDP Task id */
volatile int udpresult;    /* Result passed to task */
struct knx_udpmsg *udpmsgp; /* Saved UDP message pointer */

/* Application UDP callback function */

int myUDPFn(struct knx_udpmsg *msgp, void *userp)
{
    if ((long)userp != 0xFEEDFACEL)
        return (-1);      /* Not my UDP datagram */

    udpmsgp = msgp;      /* Save the message pointer */
    udpresult = 0;      /* Got a response */
    cjtkwake(udptaskid); /* Let UDP Task resume */
    return (0);          /* Accept the message */
}

/* Application task which sends and receives UDP datagrams*/

void myUDPTask(void)
{
    struct in_addr fhost; /* Foreign host IPv4 address */
    unsigned long handle; /* UDP channel handle */
    char *dp;             /* Data pointer */

    udptaskid = cjtkid(); /* Provide my task id */
    kn_inet_addr("192.168.5.12", &fhost); /* Destination */
    kn_inet_addr("192.168.5.62", &lhost); /* Local address */
    if ( (handle = kn_udpopen(&fhost,
                             45, /* Foreign port */
                             &lhost, /* Local host */
                             43, /* Local port */
                             myUDPFn, (void *)0xFEEDFACEL)) == 0)
        return; /* Cannot open UDP channel */

    udpresult = -1;
    dp = "KwikNet is asking for a UDP response.\n";
    if (kn_udpsend(handle, &fhost, 45, dp, strlen(dp)) == 0)
    {
        /* Wait 60 seconds for response */
        if (udpresult == -1) /* Reply may be here already */
            cjtkwaitm(cjtmconvert(60000L));
        if (udpresult == 0)
        {
            kn_dprintf(0, "Got UDP reply: %s.\n",
                      udpmsgp->xudpm_datap);
            kn_udpfree(udpmsgp); /* Free the message */
        }
    }
    kn_udpclose(handle); /* Close the UDP channel */
}
```

Purpose **Send a UDP Datagram on a Network****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_API.H*.

```
#include "KN_LIB.H"
int kn_udpsend(unsigned long udpchannel,
                struct in_addr *fhostp, int fport,
                char *bufp, int length);
```

Description *Udpchannel* is a UDP handle acquired with a previous call to *kn_udpopen()*.

Fhostp is a pointer to a structure containing the IPv4 address, in net endian form, of the foreign host to whom you wish to send a UDP datagram. The BSD structure *in_addr* is defined in Treck header file *TRSOCKET.H* as follows:

```
struct in_addr {
    u_long s_addr;           /* IP address (net endian) */
};
```

Fport is the foreign port number to which you are sending the UDP datagram.

Bufp is a pointer to a character buffer containing the data bytes which you wish to send in the UDP datagram.

Length is the number of bytes in the character buffer referenced by *bufp*.

Returns If successful, a value of 0 is returned.
 On failure, the error status -1 is returned.

The most likely reasons for failure are:
 Invalid UDP channel handle
 Destination is not accessible via any local network interface
 No memory is available to construct and send the datagram
 Length exceeds the maximum supported UDP/IP data segment size.

Example See example in the description of *kn_udpopen()*.

See Also *kn_udpopen()*

Purpose	Yield to the KwikNet Task
Used by	<input checked="" type="checkbox"/> Task <input type="checkbox"/> ISP <input type="checkbox"/> Timer Procedure <input type="checkbox"/> Restart Procedure <input type="checkbox"/> Exit Procedure
Setup	Prototype is in file <i>KN_API.H</i> . <pre>#include "KN_LIB.H" int kn_yield(void);</pre>
Description	Your single threaded application must regularly yield to the KwikNet Task. Failure to yield at least at the defined KwikNet clock frequency may result in poor performance of the TCP/IP stack.
Returns	Error status is returned. The value 1 is returned if the KwikNet Task executes successfully. The value 0 is returned if the request to execute the KwikNet Task fails. The return value of 0 indicates that the KwikNet Task cannot be executed for some reason. For example, if an application UDP callback function calls <i>kn_yield()</i> when it is executed by the KwikNet Task, the call will fail because the KwikNet Task cannot execute recursively.
Restriction	Must only be called in a single threaded system by the App-Task while executing in the user domain.
See Also	<i>kn_addserver()</i>

Purpose Convert Between Network and Host Endian Forms

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup The macro definitions are in file *KN_LIB.H*.
C dependent, in-line assembly language expansions are in file *KNZZZCC.H*.
#include "KN_LIB.H"

Convert 32-bit values

netlong = *htonl(hostlong)*
hostlong = *ntohl(netlong)*

Convert 16-bit values

netshort = *htons(hostshort)*
hostshort = *ntohs(netshort)*

Description *Hostlong* is any 32-bit value in host endian form.
Netlong is any 32-bit value in net endian form.
Hostshort is any 16-bit value in host endian form.
Netshort is any 16-bit value in net endian form.

If the KwikNet Library has been configured for big endian operation, these macros do nothing since the input values require no conversion.

If the KwikNet Library has been configured for little endian operation, these macros may expand to a function call, an in-line function expansion or a series of C statements, depending upon which C compiler is being used.

The goal is always to ensure the fastest possible execution of these frequently encountered macros. When possible, these macros have been implemented using in-line assembly language statements generated by the C compiler. In some cases, the macros generate calls to assembly language functions of a form supported by the C compiler. As a last resort, the macros expand to a series of in-line C statements.

Returns The input value converted to opposite endian form.

Restriction These macros can introduce side effects. Therefore, the macro parameters must not use expressions which include operators such as *--* or *++* since they always produce side effects. You must also avoid using expressions which include function calls to fetch parameters if the functions can introduce side effects.

Example See examples in the descriptions of *kn_cksum()*, *kn_dprintf()* and *kn_fmt()*.

5. KwikNet TCP/IP Sockets

5.1 Introduction to KwikNet Sockets

Sockets is an application programming interface (API) which was developed for UNIX during the early 1980s at University of California, Berkeley. It is primarily used today for TCP programming. Dozens of books and tutorials are available for sockets programming, one of the compelling arguments for the use of sockets.

Programmers new to sockets may observe that the sockets API seems unnecessarily complex. The reasons are historical. Sockets were initially developed to allow interprocess communication via streaming devices in UNIX environments. One process would write to a connection socket and another process would read from a socket at the other end of the connection. Sockets were meant to be a general solution for all types of data transfer. On many UNIX systems you can actually pass a socket to the file *read()* and *write()* calls in place of a file descriptor.

When the Berkeley researchers wanted to extend the endpoints of the socket beyond the host system so that processes on two separate systems could talk, they implemented TCP (as well as other protocols) under the sockets API. The sockets API had to be extended to indicate the type of service to be provided by the socket. The *PF_INET* (as opposed to *PF_UNIX*) parameter in the *socket()* call is a vestige of this legacy.

This use of TCP as a carrier for sockets was TCP's first major popular application outside of the DARPA projects where it was developed. So in a very real sense, TCP owes its widespread popularity today to Berkeley UNIX and sockets.

Over the years, many simpler, cleaner TCP APIs have been proposed. However, by the time TCP became popular on non-UNIX platforms, it was too late. Programmers had become accustomed to the API and sockets had become the de-facto standard for TCP programming.

KwikNet Procedure Descriptions

Each procedure in the KwikNet TCP/IP sockets API is described in Chapter 5.4. The format of each description is identical to that presented in Chapter 4.6 and used to describe the low level services in the KwikNet Library.

However, since the KwikNet Sample Program provides a complete working illustration of how to use each of the KwikNet TCP/IP procedures, examples are not repeated in the TCP/IP sockets API descriptions presented in Chapter 5.4.

KwikNet Sockets API

The KwikNet TCP sockets API is a subset of that available on UNIX systems. Examples of networking code from other sockets-based systems can be expected to port easily. Furthermore, most of the reference material in books and tutorials will also apply to the KwikNet API.

The KwikNet API has been designed for best use in embedded systems where execution speed, code size and ease of use are of paramount importance. For these reasons, the KwikNet sockets API differs from the Berkeley sockets API. However, a standard sockets API is provided with KwikNet and can be used if code compatibility is of utmost concern.

The API differences are minor. All of the procedure names in the KwikNet TCP/IP Stack are of the form *kn_XXXXX()*. For example, Berkeley procedure *socket()* is KwikNet procedure *kn_socket()*. This naming convention has been adopted by KADAK to avoid any conflicts with symbols in your application or your C run-time libraries.

The standard sockets API uses the procedure *close()* to close a socket. A KwikNet procedure with this name would conflict with procedure *close()* in the standard C library. Consequently, the KwikNet procedure *kn_close()* must be used to close a socket.

The KwikNet sockets API is defined as a set of C macros which map the KwikNet API names to the underlying standard sockets API in the Turbo Treck TCP/IP Stack. To use the KwikNet sockets API, include the following statement in your source modules:

```
#include "KN SOCK.H"
```

To use the Berkeley standard sockets API, review the macro definitions in header file *KN SOCK.H* and call the equivalent Treck sockets functions directly. The Treck functions use the BSD function names but may require some casting of parameter types to match the function prototypes. Be sure to use *kn_close()* and *kn_errno()* (or their Treck equivalents) instead of the BSD *close()* and *errno()*.

Socket Addresses

The endpoint of a socket connection is identified using a socket address. The specification of a socket address is dependent upon the protocol used for communication. The sockets API uses a generalized *sockaddr* structure to specify a socket address. When using the TCP or UDP protocols, it is convenient to cast each *sockaddr* pointer to reference the following Internet specific socket address structure:

```
struct sockaddr_in {
    unsigned short  sin_family;      /* Address family = AF_INET          */
    unsigned short  sin_port;        /* TCP, UDP: protocol port          */
    struct in_addr  sin_addr;        /* TCP, UDP: IP address              */
    char            sin_zero[8];     /* TCP, UDP: unused (0)              */
};
```

Member *sin_family* always specifies the Internet Protocol address family (*AF_INET*), stored in host endian form. The protocol port number is stored in member *sin_port* in net endian form. The protocol IP address is stored in member *sin_addr.s_addr* in net endian form. The array *sin_zero[]* is unused and is ignored by KwikNet.

Non-Blocking Sockets

When operations are performed using a socket, the caller requesting the action is usually forced to wait until the operation completes. The socket is said to be in blocking mode.

KwikNet offers an alternative mode of operation in which the socket action is allowed to proceed without blocking the caller. In this case, the socket is said to be in non-blocking mode.

When a socket is created it is placed in blocking mode. Thereafter, your application can call the sockets procedure `kn_setsockopt()` to alter the operating mode of the socket. KwikNet socket option `SO_NONBLOCK` can be used to place the socket in non-blocking mode. The same option can be used to restore the socket to blocking mode. Since `SO_NONBLOCK` is a unique KwikNet option, it is flagged as non-standard in the summary of sockets options presented in Chapter 5.3.

KwikNet Error Codes

KwikNet socket procedures return a positive value or 0 if the call is successful or -1 if the call fails. Procedures which return a socket descriptor return a non-negative value if the call is successful.

Standard sockets procedures also return additional error information in the UNIX global variable `errno`. This is the same variable which many standard C libraries use to save error results. However, KwikNet must coexist with these libraries. Furthermore, since KwikNet must operate in multitasking environments, it cannot use `errno` since the results of operations by different tasks would be indeterminate.

For these reasons, global variable `errno` is not altered by KwikNet. Instead, the completion status of each socket operation is recorded within the socket descriptor upon completion of the operation. This error status is then made available to the caller through the `kn_errno()` procedure. Be sure to use `kn_errno()` (or its Treck equivalent) instead of the BSD `errno()` function.

Unfortunately, procedures `kn_socket()` and `kn_select()` have no socket descriptor in which to save the error code. Hence, you cannot use `kn_errno()` to retrieve error information after calls to these procedures.

Error codes returned by `kn_errno()` are integer values which are a subset of the standard Berkeley error codes. These error codes are summarized in Appendix B.

5.2 Socket Types

Procedure *kn_socket()* is used to create a socket. The KwikNet TCP/IP Stack only supports sockets for use in the communication domain which uses the protocol family known as the ARPA Internet Protocol, identified as *PF_INET*.

Two socket types are supported: type *SOCK_STREAM* for use with the TCP protocol and type *SOCK_DGRAM* for use with the UDP protocol.

Stream Socket (for TCP)

A socket of type *SOCK_STREAM* provides a sequenced, reliable, full duplex connection between two end points. The local end point is identified using a *kn_bind()* call. A stream socket must be in a connected state before any data can be sent or received using it. A connection is created with a *kn_connect()* call which identifies the remote end of the connection. Once connected, data may be transferred using procedures *kn_send()* and *kn_recv()*. Out-of-band data can be transferred using the *MSG_OOB* option in either of these calls. Finally, when a session has been completed and the local socket is no longer required, procedure *kn_close()* can be used to delete the socket.

The communications protocols used to implement a *SOCK_STREAM* ensure that data is not lost or duplicated. If a piece of data accepted by the protocol stack cannot be successfully delivered within a reasonable length of time, then the connection is considered broken. The *kn_send()* or *kn_recv()* procedures will fail and the socket will report an error code when subsequently interrogated with a *kn_errno()* call. Some protocols include options to keep sockets *warm* by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period of time.

Functions *kn_sendto()*, *kn_writev()*, *kn_recvfrom()* and *kn_readv()* can also be used to send and receive data once a connection has been established. Since the end points of a connected socket are known, there is no need to provide the destination address or storage for the source address when using these functions.

Datagram Socket (for UDP)

A socket of type *SOCK_DGRAM* provides a connectionless, unreliable method for delivering messages of a fixed, usually small, maximum length. The messages are called datagrams. Although the local end point can be identified using a *kn_bind()* call, it is not necessary to bind the socket before using it. A *SOCK_DGRAM* socket allows a datagram to be sent to a correspondent named in the *kn_sendto()* call. Datagrams are received using the *kn_recvfrom()* procedure which identifies the address from which the data is received. Finally, when a session has been completed and the local socket is no longer required, procedure *kn_close()* can be used to delete the socket.

Although a socket of type *SOCK_DGRAM* is connectionless, the *kn_connect()* procedure can still be used to identify the specific peer with whom a conversation is to be held. The connection defines the address to which datagrams are to be sent and the only address from which datagrams are to be received. Furthermore, *kn_connect()* can be called at any time to change the connection address.

Functions *kn_writev()* and *kn_readv()* can also be used to send and receive data. If you have provided a destination address with a *kn_connect()* call, you can use function *kn_send()* to send data to that destination. And you can always use function *kn_recv()* to receive data, as long as you do not need to identify the source.

The maximum size of a UDP datagram that KwikNet can send or receive is dictated by a number of factors including memory availability, buffering capability at the IP protocol layer, IP fragmentation limitations, network interface restrictions on datagram size and routing effects as the UDP datagram finds its way to the intended destination. If you send a UDP datagram which exceeds 576 bytes (512 bytes of UDP data), it becomes your responsibility to ensure that the receiving host and intervening routers can handle such UDP datagrams. When receiving, your application must be able to accept the largest expected UDP datagram or risk loss of data.

Using UDP Sockets

Special care must be taken when using a socket of type *SOCK_DGRAM* with a connectionless protocol such as UDP. When a UDP socket is created, it is considered bound to local IP address 0 and port 0. The newly created socket is idle, unable to receive data from any foreign host.

The UDP socket remains idle, bound to local IP address 0 and port 0, until you explicitly bind it otherwise using *kn_bind()* or until you send data. When data is sent using a UDP socket whose local port is still 0, a unique non-zero port number is automatically bound to the socket to identify the source of the datagram. The local non-zero port number identifies the source in all subsequent transmissions. Once the port number is known, the socket can receive datagrams directed to that port.

Reception is governed by the local IP address and port to which the UDP socket is bound. As long as the socket is bound to IP address 0, the socket can receive datagrams arriving on any of the local interfaces. Once bound to a specific IP address, the socket can only receive datagrams directed to that address. In either case, the datagrams will not be delivered to the socket unless they are destined for the port to which the socket is bound.

Transmission is governed by the destination to which the UDP socket is connected. A socket can be connected to a particular foreign address and port using function *kn_connect()*. Once connected, any of the socket send functions, including *kn_send()*, can be used to send a datagram to the connected host without having to explicitly identify the destination. However, a socket does not have to be connected prior to sending a datagram. Whether connected or not, functions *kn_sendto()* and *kn_writev()* can always be used to send a datagram to a specific foreign host.

The UDP socket connection also restricts the datagrams which the socket can receive. Once the socket has been connected to a specific foreign host, the socket will only receive datagrams directed to it from that foreign host.

Unless you specifically bind a UDP socket to a particular port, a unique port number will be automatically assigned to the socket the first time the socket is connected or a datagram is sent via the socket. Normally, KwikNet will not allow you to bind a socket to a particular port number if that port number is already used by another UDP channel. To overcome this restriction, you can set the socket *SO_REUSEADDR* option to allow the port number to be reused.

UDP Sockets Examples

The following examples illustrate how a UDP socket can be used in a variety of circumstances for different purposes.

Example 1

To use a UDP socket to communicate on a specific network with one and only one foreign host, proceed as follows. Create a UDP socket and use *kn_bind()* to bind it to a specific local IP address and port. Then call *kn_connect()* to establish a logical connection with the foreign host's IP address and port. Use *kn_recv()* and *kn_send()* to communicate using the socket.

Example 2

A variation of Example 1 will illustrate how logical connections can be manipulated. Until a connection is established, function *kn_send()* cannot be used to send a datagram. However, either *kn_sendto()* or *kn_writev()* can be used at any time to send a datagram to any foreign host without affecting the logical connection. It should also be noted that function *kn_connect()* can also be called to change the connection. The connected foreign host is always used as the destination if a foreign address is not explicitly provided in a request to send a datagram.

Example 3

To accept datagrams from only the foreign hosts to whom you send a datagram, proceed as follows. Create a UDP socket and send a datagram to a foreign host using *kn_sendto()* or *kn_writev()*. The socket is immediately given a unique port number which will be known only to the foreign hosts with which you communicate. Your socket will receive datagrams from any local network interface provided they are directed to your particular port. Note that you cannot use function *kn_send()* to send datagrams on this socket since the socket is not logically connected.

Note

KwikNet provides an alternate method of sending and receiving UDP datagrams using the UDP channel API described in Chapter 4.1. However, if you are familiar with the use of the sockets API with sockets of type *SOCK_DGRAM*, there is no compelling reason not to use UDP sockets.

5.3 Socket Options

The operation of sockets is controlled by socket level options. Options are always present at the socket level identified as *SOL_SOCKET*. KwikNet also supports TCP options at protocol level *IPPROTO_TCP*. There are no UDP protocol options. The KwikNet options are listed in file *KN_SOCK.H* but are defined in Treck file *TRSOCKET.H*. Procedures *kn_getsockopt()* and *kn_setsockopt()* are used to access and modify these options.

The following commonly used options are recognized by KwikNet. Options marked R can be read using *kn_getsockopt()*. Options marked W can be modified using *kn_setsockopt()*. Unmarked options will return an error indication if referenced. The options marked with > are non-standard extensions offered only by KwikNet.

In addition to these commonly used options, a number of other socket, TCP and IP options are provided by Treck to adjust various operating characteristics. Refer to the Treck TCP/IP User Manual and to Treck file *TRSOCKET.H* for details.

Option	type	UDP	TCP	Purpose
<i>SO_REUSEADDR</i>	<i>bool</i>	RW	RW	Local address reuse
<i>SO_ACCEPTCONN</i>	<i>bool</i>		R	Check if socket is a listening socket
<i>SO_KEEPAIVE</i>	<i>bool</i>		RW	Keep connections alive
<i>SO_DONTROUTE</i>	<i>bool</i>			Routing bypass for outgoing messages
<i>SO_BROADCAST</i>	<i>bool</i>	RW		Permission to transmit broadcast messages
<i>SO_OOBINLINE</i>	<i>bool</i>		RW	Allow out-of-band data in band
<i>SO_LINGER</i>	<i>struct</i>		RW	Linger on close if data present
<i>SO_SNDBUF</i>	<i>u/s long</i>		RW	Buffer size for send
<i>SO_RCVBUF</i>	<i>u/s long</i>		RW	Buffer size for receive
<i>SO_SNDLOWAT</i>	<i>u/s long</i>		RW	Buffer low limit for send
<i>SO_RCVLOWAT</i>	<i>u/s long</i>		RW	Buffer low limit for receive
<i>SO_SNDTIMEO</i>	<i>struct</i>			Timeout limit for send
<i>SO_RCVTIMEO</i>	<i>struct</i>			Timeout limit for receive
<i>SO_ERROR</i>	<i>int</i>	R	R	Get and clear error on a socket
<i>SO_NONBLOCK</i>	<i>bool</i>	RW	RW	> Socket operates in non-blocking mode
<i>TCP_NODELAY</i>	<i>bool</i>		RW	Do not delay data send to coalesce data
<i>TCP_NOPUSH</i>	<i>bool</i>		RW	Do not push last byte of data sent
<i>TCP_MAXSEG</i>	<i>int</i>		RW	Get maximum segment size (MSS)

Option *SO_REUSEADDR* indicates that the rules used in validating addresses supplied in a *kn_bind()* call should allow reuse of local addresses.

Option *SO_ACCEPTCONN* can be used to determine if a socket is a listening socket, a socket which accepts requests for connection.

Option *SO_KEEPAIVE* enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken. A process attempting to read from or write to the socket receives an error indication.

Option *SO_DONTROUTE* indicates that outgoing messages should bypass the standard routing facilities. KwikNet does not support this option.

Option *SO_BROADCAST* requests permission to send broadcast datagrams on the socket.

Option *SO_OOBINLINE*, used with protocols that support out-of-band data, requests that out-of-band data be placed in the normal data queue as it is encountered. The data will then be accessible with the *kn_recv()* call without the need to specify the *MSG_OOB* flag.

Option *SO_LINGER* controls the action taken when unsent messages remain queued on a socket at the time a *kn_close()* request to delete the socket is made. If the socket promises reliable delivery of data and *SO_LINGER* is set, KwikNet will block the caller on the *kn_close()* attempt until it is able to successfully transmit the data or until it decides it is unable to do so. A timeout period, termed the linger interval, is specified in the *kn_setsockopt()* call at the time option *SO_LINGER* is enabled. If *SO_LINGER* is disabled at the time that a *kn_close()* request is issued, KwikNet will process the close in a manner that allows the caller to resume as quickly as possible.

Options *SO_SNDBUF* and *SO_RCVBUF* are used to adjust the normal buffer sizes allocated for send and receive buffers respectively. The buffer size may be increased for high volume connections or may be decreased to limit the possible backlog of incoming data. These values reflect the total amount of data which can be buffered at the socket. The receive buffer size determines the largest TCP window size which the socket will advertise. The send buffer size determines the maximum number of bytes which can be held in the socket pending acknowledgment of receipt by the peer. When a socket is created, KwikNet sets both buffering limits to 8 Kbytes.

Options *SO_SNDLOWAT* and *SO_RCVLOWAT* set the minimum data count for send and receive operations respectively.

Options *SO_SNDTIMEO* and *SO_RCVTIMEO* set a timeout value for send and receive operations respectively. KwikNet does not support timeouts for send or receive.

Option *SO_ERROR* can be used to fetch the most recent error code recorded in the socket. The socket's error code is then reset (cleared). This option is useful for checking for asynchronously occurring socket errors.

Non-Standard Socket Options

The non-standard option *SO_NONBLOCK* can be used to set a socket into non-blocking mode so that the socket user will not be forced to wait if a requested operation cannot be completed at the time of the request. It can also be used to restore a socket to blocking mode so that the socket user will be forced to wait until the requested operation completes. This option's integer parameter is non-zero for non-blocking mode or zero for blocking mode.

TCP Protocol Options

Option *TCP_NODELAY* can be used to adjust the way the TCP protocol sends data. Normally, data is allowed to collect in the socket until a reasonable packet of data can be delivered to the peer. The data threshold is determined by the receive window size announced by the peer. This throttling mechanism assures reasonable delivery times without large numbers of small packets. The *TCP_NODELAY* option can be used to override this mechanism and force data to be sent whenever it is available.

Option *TCP_NOPUSH* can be used to prevent the setting of the TCP push flag *PSH* when data is transmitted. In this case, the data will not be pushed up to the receiving application just because the sender is unable to provide data fast enough to keep the data stream filled.

Option *TCP_MAXSEG* can be used to determine the socket's maximum segment size (MSS) for transmission.

This page left blank intentionally.

5.4 KwikNet Socket Services

KwikNet Socket Service Summary

The following list summarizes all of the KwikNet sockets procedures which are accessible to the user. They are grouped functionally for easy reference.

<i>kn_socket</i>	Create a socket (an endpoint for communication)
<i>kn_bind</i>	Bind a local address to a socket
<i>kn_connect</i>	Connect a socket to a specific address
<i>kn_listen</i>	Request a socket to listen for connection requests
<i>kn_accept</i>	Accept a connection request and establish a new socket
<i>kn_close</i>	Close a socket
<i>kn_recv</i>	Receive data from a connected socket
<i>kn_recvfrom</i>	Receive data from a socket (gets sender's address)
<i>kn_readv</i>	Receive scattered data from a socket (gets sender's address)
<i>kn_send</i>	Send data to a socket
<i>kn_sendto</i>	Send data to a socket (with destination address)
<i>kn_writev</i>	Send scattered data to a socket (with destination address)
<i>kn_errno</i>	Fetch most recent status result (error) recorded for a socket
<i>kn_shutdown</i>	Shut down all or part of a full duplex socket connection
<i>kn_select</i>	Select sockets ready to receive or send data
<i>kn_getpeername</i>	Get the address of the remote end of a connected socket
<i>kn_getsockname</i>	Get the local address of a socket
<i>kn_getsockopt</i>	Get a particular socket option
<i>kn_setsockopt</i>	Set a particular socket option

The following BSD-like services, available from the KwikNet Library, are also of use when programming applications which use UDP or TCP as their protocol.

<i>netlong</i>	<code>= htonl(hostlong)</code>	Convert <i>long</i> from host to network endian form
<i>netshort</i>	<code>= htons(hostshort)</code>	Convert <i>short</i> from host to network endian form
<i>hostlong</i>	<code>= ntohl(netlong)</code>	Convert <i>long</i> from network to host endian form
<i>hostshort</i>	<code>= ntohs(netshort)</code>	Convert <i>short</i> from network to host endian form

This page left blank intentionally.

Purpose **Accept a Connection Request****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_accept(int s, struct sockaddr *addr, int *addrlen);
```

Description *s* is a socket descriptor identifying the socket on which to wait. The socket *s* must have been created with a call to *kn_socket()*, bound to an address with a call to *kn_bind()* and then made ready to receive connection requests with a call to *kn_listen()*.

Addr is a pointer to storage for the address of the client making the request for connection. The format of the IP address in structure *sockaddr* is described in header file *KN_SOCK.H*.

Addrlen is a pointer to storage for the length of the client address. On entry, the integer at **addrlen* must define the maximum storage available within the structure referenced by *addr*.

If a request for connection is pending at socket *s*, *kn_accept()* creates a new socket with the same properties as socket *s* and returns the socket descriptor of the new socket to the caller. The new socket must be used for data transfers to or from the client to which the new socket is connected. The new socket cannot be used to accept more connections.

In most cases, the *kn_accept()* caller will be forced to wait for a connection request if none is pending at the time of the call. However, if the socket *s* was marked as non-blocking, *kn_accept()* will return immediately to the caller with an error indication if an outstanding connection request is not present. Socket *s* remains open listening for connection requests.

Returns If successful, a non-negative socket descriptor is returned.
The structure at **addr* contains the client's address.
The length of the address (in bytes) is stored at **addrlen*.

On failure, the error status *-1* is returned. The storage at **addr* and **addrlen* is unaltered.

...more

Returns ...continued

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	Parameter <i>addr</i> or <i>addrlen</i> is invalid or parameter <i>*addrlen</i> specifies a length that is less than that required to accommodate a valid address or the socket is no longer accepting connections.
<i>ECONNABORTED</i>	The connection was aborted.
<i>ENOBUFS</i>	Memory needed to service the request is unavailable.
<i>EOPNOTSUPP</i>	The referenced socket is not of type <i>SOCK_STREAM</i> .
<i>EPERM</i>	Socket descriptor <i>s</i> is not a listening socket. You must call <i>kn_listen()</i> before calling <i>kn_accept()</i> .
<i>EWouldBlock</i>	The socket is marked non-blocking and no requests for connection are present to be accepted.

See Also *kn_bind()*, *kn_listen()*, *kn_socket()*

Purpose Bind a Local Address to a Socket**Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_bind(int s, struct sockaddr *localaddr, int addrlen);
```

Description *s* is a socket descriptor identifying the socket to be bound.*Localaddr* is a pointer to a structure containing the local address to which the socket must be bound. The format of the IP address in structure *sockaddr* is described in header file *KN_SOCK.H*.*Addrlen* is the length of the address at **localaddr*, measured in bytes.**Returns** If successful, a value of 0 is returned.
On failure, the error status -1 is returned.The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EADDRINUSE</i>	The specified address is already in use.
<i>EINVAL</i>	The socket is already bound to an address or parameter <i>localaddr</i> or <i>addrlen</i> is invalid.

Note When a socket of type *SOCK_DGRAM* is bound to the address identified by *localaddr*, the socket is immediately primed to receive datagrams directed to that address from any foreign source.**See Also** *kn_accept()*, *kn_getsockname()*, *kn_listen()*, *kn_socket()*

Purpose **Close a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_close(int s);
```

Description *s* is a socket descriptor identifying the socket to be closed.

When a socket is closed, associated naming information and queued data are discarded.

Returns If successful, a value of 0 is returned.

On failure, the error status *-1* is returned.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EALREADY</i>	The socket is already in the process of closing.
<i>ETIMEDOUT</i>	The socket's linger option was enabled with a non-zero timeout value. The linger timeout expired before the TCP close handshake with the remote host could be completed (blocking TCP socket only).

See Also *kn_socket()*

Purpose **Connect a Socket to a Specific Address****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_connect(int s, struct sockaddr *destaddr, int addrlen);
```

Description *s* is a socket descriptor identifying the socket to which the specified address is to be bound.

Destaddr is a pointer to a structure containing the specific address to which the socket must be bound. The format of the IP address in structure *sockaddr* is described in header file *KN_SOCK.H*.

Addrlen is the length of the address at **destaddr*, measured in bytes.

If socket *s* is of type *SOCK_STREAM*, then an attempt will be made to establish a connection with the address specified in the call. Stream sockets may successfully connect only once.

If socket *s* is of type *SOCK_DGRAM*, then **destaddr* specifies the address of the peer with which the socket is to be associated. This is the address to which datagrams are to be sent and is the only address from which datagrams are to be received. For datagram sockets, you can call *kn_connect()* at any time to change the connection address. To disconnect a datagram socket, call *kn_connect()* passing it a null (all zeroes) address in **destaddr*.

Returns If successful, a value of 0 is returned.
On failure, the error status -1 is returned.

...more

Returns

...continued

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	The socket is already bound to an address or parameter <i>destaddr</i> or <i>addrlen</i> is invalid.
<i>EADDRNOTAVAIL</i>	The specified address is not available from the local machine or a broadcast address is not allowed as a destination for this socket.
<i>EADDRINUSE</i>	The specified address is already in use.
<i>EAFNOSUPPORT</i>	Addresses in the address family specified by <i>*destaddr</i> cannot be used with this socket.
<i>ETIMEDOUT</i>	Timed out before connection could be established. (see note below)
<i>ECONNREFUSED</i>	The attempt to connect was forcefully rejected. (see note below)
<i>EHOSTUNREACH</i>	There is no route available to reach the destination address specified. (see note below)
<i>EINPROGRESS</i>	The socket is non-blocking and the connection has not yet been established.
<i>EALREADY</i>	The socket is non-blocking and a previous connection attempt is in progress but is not yet complete.
<i>EPERM</i>	The socket descriptor <i>s</i> is a listening socket. Do not call <i>kn_connect()</i> after calling <i>kn_listen()</i> .
<i>ENOBUFS</i>	Memory needed to service the request is unavailable.
** note **	You should close the socket identified by descriptor <i>s</i> to ensure that the connection attempt is aborted. Then, if necessary, call <i>kn_socket()</i> to obtain a new socket descriptor and attempt the connection again.

See Also

kn_bind(), *kn_getsockname()*, *kn_select()*, *kn_socket()*

Purpose **Get Error Code from Recent Socket Operation****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_errno(int s);
```

Description *s* is a socket descriptor identifying the socket from which error information is to be retrieved.**Returns** If successful, a positive error code is returned. These error codes are summarized in Appendix B. The error code identifies the reason for the failure, if any, of the most recent operation attempted using socket *s*.

An error status of *EBADF* is returned if the socket descriptor *s* is invalid, precluding the interrogation of the socket.

Usually the error code retrieved from socket *s* corresponds to the result of the most recent sockets call made by your application. However, since KwikNet always records any relevant socket error which it detects, your call to *kn_errno()* may actually retrieve an error code recorded by KwikNet at some time after your socket operation.

For example, the following error codes, if observed, could indicate that the corresponding error on the socket referenced by descriptor *s* was detected by KwikNet asynchronously to your application.

<i>ESHUTDOWN</i>	The connection has been shut down.
<i>ECONNRESET</i>	The connection has been reset.

Note You can use this procedure to advantage to interrogate the status of a listening socket while it continues to operate.**Note** The error code associated with socket *s* remains unaltered. To read the error code and reset the error code to 0, call *kn_getsockopt()* with option *SO_ERROR*.**See Also** *kn_getsockopt()*

Purpose **Get the Address (Name) of the Remote End of a Connected Socket**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_getpeername(int s, struct sockaddr *destaddr,
                  int *addrlen);
```

Description *s* is a socket descriptor identifying the socket for which the remote end (peer) address is desired.

Destaddr is a pointer to storage for the address of the remote end of the current connection on socket *s*. The format of the IP address in structure *sockaddr* is described in header file *KN_SOCK.H*.

Addrlen is a pointer to storage for the length of the remote address. On entry, the integer at **addrlen* must define the maximum storage available within the structure referenced by *destaddr*.

Returns If successful, a value of 0 is returned.
The structure at **destaddr* contains the remote end address.
The storage at **addrlen* is unaltered.

On failure, the error status -1 is returned. The storage at **destaddr* and **addrlen* is unaltered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOTCONN</i>	The socket is not connected.
<i>EINVAL</i>	Parameter <i>destaddr</i> or <i>addrlen</i> is invalid or parameter <i>*addrlen</i> specifies a length that is less than that required to accommodate a valid address.

See Also *kn_connect()*, *kn_getsockname()*, *kn_socket()*

Purpose **Get the Local Address (Name) of a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_getsockname(int s, struct sockaddr *localaddr,
                  int *addrlen);
```

Description *s* is a socket descriptor identifying the socket for which the local address is desired.

Localaddr is a pointer to storage for the local address assigned to socket *s*. The format of the IP address in structure *sockaddr* is described in header file *KN_SOCK.H*.

Addrlen is a pointer to storage for the length of the local address. On entry, the integer at **addrlen* must define the maximum storage available within the structure referenced by *localaddr*.

Returns If successful, a value of 0 is returned.
The structure at **localaddr* contains the local address.
The storage at **addrlen* is unaltered.

On failure, the error status -1 is returned. The storage at **localaddr* and **addrlen* is unaltered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	Parameter <i>localaddr</i> or <i>addrlen</i> is invalid or parameter <i>*addrlen</i> specifies a length that is less than that required to accommodate a valid address.

See Also *kn_bind()*, *kn_getpeername()*, *kn_socket()*

Purpose **Get a Particular Socket Option****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_getsockopt(int s, int level, int optionid,
                  void *optionval, int *optionlen);
```

Description *s* is a socket descriptor identifying the socket for which the particular socket information is desired.

Level is an identifier indicating the socket or protocol level for which information is required. Use *SOL_SOCKET* for the highest, socket level options. Use *IPPROTO_TCP* for TCP protocol level options. The Turbo Treck TCP/IP Stack also supports the use of *IPPROTO_IP* for IP protocol level options. Other protocol levels are not supported.

Optionid identifies the option of interest to the caller. The following option identifiers can be used to determine the state of the option or to retrieve its associated parameter. These options are described in Chapter 5.3. Options marked > are non-standard KADAK extensions. Only one option can be specified in each call. The Turbo Treck TCP/IP Stack supports a number of additional socket, TCP and IP options which are described in the Treck TCP/IP User Manual.

<i>SO_REUSEADDR</i>	<i>bool</i>	UDP	TCP	Local address reuse
<i>SO_ACCEPTCONN</i>	<i>bool</i>		TCP	Check for listening socket
<i>SO_KEEPAIVE</i>	<i>bool</i>		TCP	Keep connections alive
<i>SO_BROADCAST</i>	<i>bool</i>	UDP		Permission to broadcast messages
<i>SO_OOBINLINE</i>	<i>bool</i>		TCP	Allow out-of-band data in band
<i>SO_LINGER</i>	<i>struct</i>		TCP	Linger on close if data present
<i>SO_SNDBUF</i>	<i>u/s long</i>		TCP	Buffer size for send
<i>SO_RCVBUF</i>	<i>u/s long</i>		TCP	Buffer size for receive
<i>SO_SNDLOWAT</i>	<i>u/s long</i>		TCP	Buffer low limit for send
<i>SO_RCVLOWAT</i>	<i>u/s long</i>		TCP	Buffer low limit for receive
<i>SO_ERROR</i>	<i>int</i>	UDP	TCP	Get and clear error on a socket
<i>SO_NONBLOCK</i>	<i>bool</i>	UDP	TCP	> Socket is non-blocking mode
<i>TCP_NODELAY</i>	<i>bool</i>		TCP	Do not delay send to coalesce data
<i>TCP_NOPUSH</i>	<i>bool</i>		TCP	Do not push last byte of data sent
<i>TCP_MAXSEG</i>	<i>int</i>		TCP	Get maximum segment size (MSS)

...more

Description ...continued

Optionval is a pointer to storage for the option value. The size of each option is indicated in the option list. Note that *bool* is an *int* result that is *1* if the option is enabled or *0* if the option is disabled. The type *u/s long* indicates a 32-bit value usually represented by an *unsigned long*. The structure *linger* required by option *SO_LINGER* is described in header file *KN_SOCK.H*.

Optionlen is a pointer to storage for the length of the returned option value. On entry, the integer at **optionlen* must define the maximum storage available at the location referenced by *optionval*.

Returns

If successful, a value of *0* is returned.

The storage at **optionval* contains the option value.

The length of the option value (in bytes) is stored at **optionlen*.

On failure, the error status *-1* is returned. The storage at **optionval* and **optionlen* is unaltered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>ENOPROTOOPT</i>	The option is unknown at the level indicated.
<i>EINVAL</i>	Parameter <i>optionval</i> or <i>optionlen</i> is invalid or parameter <i>*optionlen</i> specifies a length that is less than that required to accommodate the result.

See Also *kn_setsockopt()*, *kn_socket()*

Purpose Request a Socket to Listen for Connection Requests**Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_listen(int s, int backlog);
```

Description *s* is a socket descriptor identifying the socket on which requests for connection will be enqueued. The socket *s* must have been created with a call to *kn_socket()* and bound to an address with a call to *kn_bind()*. The socket must be of type *SOCK_STREAM*.*Backlog* is the maximum number of pending connection requests which the socket is permitted to queue. *Backlog* must be greater than or equal to 0 and no greater than the maximum allowed by your KwikNet Library configuration.Any request for a connection to the socket will be permitted up to the maximum specified by *backlog*. Requests are kept in a queue in the order received. When the application calls *kn_accept()*, it is given a new socket connected to the client from whom the connection request was received.

If a request for connection is received while the socket queue is full, the request is rejected. Subsequent actions, if any, will be determined by the client whose request was rejected.

Returns If successful, a value of 0 is returned.
On failure, the error status -1 is returned.The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	The connection queue specified by <i>backlog</i> is < 0.
<i>EOPNOTSUPP</i>	The operation is not supported by a socket unless the socket is of type <i>SOCK_STREAM</i> .

See Also *kn_accept()*, *kn_bind()*, *kn_socket()*

Purpose **Receive Scattered Data from a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_readv(int s, struct iovec *iovecp, int iovcnt);
```

Description *s* is a socket descriptor identifying the socket from which data is to be received. The socket must be connected.*iovecp* is a pointer to an array of data vectors describing the locations of storage buffers for the received data. Each data vector is an *iovec* structure which is described in file *KN_SOCK.H* as follows:

```
struct iovec {
    void    *iov_base;           /* Data pointer          */
    int     iov_len;            /* Length of data        */
};
```

iov_base is a pointer to storage for the received data.*iov_len* is the data buffer size, measured in bytes.*iovcnt* is an integer defining the number of data vectors (*iovec* structures) which are provided in the array referenced by parameter *iovecp*.**Returns** If successful, the total number of bytes of data received is returned.
 If the socket is closed by the sender, the value 0 is returned.
 The array of *iovec* structures at **iovecp* is unaltered.

Since the data vectors are unaltered, you can use them after the call to examine the received data.

On failure, the error status -1 is returned.

...more

Returns ...continued

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	Parameter <i>iovecp</i> or <i>iovcnt</i> is invalid or the buffer length in a data vector was declared to be less than 0 or the sum of the data vector lengths exceeds the maximum size of an integer.
<i>ENOTCONN</i>	The socket is not connected.
<i>EWOULDBLOCK</i>	The socket is marked non-blocking and no data is available for reading.
<i>ESHUTDOWN</i>	The peer has closed the connection and no data is available for reading.
<i>ENOBUFS</i>	Memory needed to service the request is unavailable.

Restriction If there is no data available at the socket, the caller will be blocked waiting for data to arrive unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a -1 error status and the error code *EWOULDBLOCK* will be stored in the socket descriptor.

Restriction On 16-bit processors, the amount of data which can be received is restricted to 32767 bytes because the value returned by *kn_readv()* is a signed integer.

See Also *kn_connect()*, *kn_recv()*, *kn_recvfrom()*,
kn_socket(), *kn_writev()*

Purpose **Receive Data from a Connected Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_recv(int s, void *buf, int len, int flags);
```

Description *s* is a socket descriptor identifying the connected socket from which data is to be received. Note that the socket must be connected either by calling *kn_connect()* to connect or by calling *kn_accept()* to accept a connected socket.*Buf* is a pointer to storage for the received data.*Len* is the buffer size, measured in bytes.*Flags* is a control variable used to modify the receive process. *Flags* is 0 or the logical OR of any of the following values.

<i>MSG_OOB</i>	Read out-of-band data.
<i>MSG_PEEK</i>	Peek at the received data but do not remove the data from the socket.
<i>MSG_DONTWAIT</i>	Receive in non-blocking fashion.

Returns If successful, the number of bytes of data stored at **buf* is returned.
If the socket is closed by the sender, the value 0 is returned.
On failure, the error status -1 is returned.The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	Parameter <i>buf</i> or <i>len</i> is invalid or the buffer length <i>len</i> is declared to be less than 0 or an error was encountered processing out-of-band data.
<i>ENOTCONN</i>	The socket is not connected.
<i>EMSGSIZE</i>	The buffer length is too small to receive the message atomically as required by the socket protocol.
<i>EWOULDBLOCK</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> and no data is available for reading.
<i>ESHUTDOWN</i>	The peer has closed the connection and no data is available for reading.
<i>ENOBUFS</i>	Memory needed to service the request is unavailable.
...more	

Restriction If there is no data available at the socket, the caller will be blocked waiting for data to arrive unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a *-1* error status and the error code *EWOULDBLOCK* will be stored in the socket descriptor.

See Also `kn_connect()`, `kn_readv()`, `kn_recvfrom()`, `kn_socket()`

Purpose **Receive Data from a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_recvfrom(int s, void *buf, int len, int flags,
                struct sockaddr *from, int *fromlen);
```

Description *s* is a socket descriptor identifying the socket from which data is to be received. The socket can be connected or unconnected.*Buf* is a pointer to storage for the received data.*Len* is the buffer size, measured in bytes.*Flags* is a control variable used to modify the receive process. *Flags* is 0 or the logical OR of any of the following values.*MSG_PEEK* Peek at the received data but do not remove the data from the socket.*MSG_DONTWAIT* Receive in non-blocking fashion.*From* is a pointer to storage for the address of the source of the received data. The format of the IP address in structure *sockaddr* is described in header file *KN_SOCK.H*.The purpose of this parameter is to allow the sender to be identified when using a connectionless socket such as that used for UDP datagrams on a socket of type *SOCK_DGRAM*. The format of the address is suitable for sending a response using procedure *kn_sendto()*. Set *from* to *NULL* if the address of the message sender is not required.*Fromlen* is a pointer to storage for the length of the sender's address. On entry, the integer at **fromlen* must define the maximum storage available within the structure referenced by *from*.

...more

Returns If successful, the number of bytes of data stored at **buf* is returned.
 If the socket is closed by the sender, the value 0 is returned.
 The structure at **from* contains the sender's address.
 The storage at **fromlen* is set to the actual size of the address.

On failure, the error status -1 is returned. The storage at **from* and **fromlen* may be altered.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	Parameter <i>buf</i> , <i>from</i> or <i>fromlen</i> is invalid or the buffer length <i>len</i> is declared to be less than 0 or parameter <i>*fromlen</i> specifies a length that is less than that required to accommodate a valid address.
<i>EMSGSIZE</i>	The buffer length is too small to receive the message atomically as required by the socket protocol.
<i>EPROTOTYPE</i>	Request is invalid for sockets using the TCP protocol.
<i>EWOULDBLOCK</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> and no data is available for reading.
<i>ENOBUFS</i>	Memory needed to service the request is unavailable.

Restriction If there is no data available at the socket, the caller will be blocked waiting for data to arrive unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a -1 error status and the error code *EWOULDBLOCK* will be stored in the socket descriptor.

Restriction This procedure cannot be used to receive from a foreign host using the TCP protocol. Use *kn_recv()* or *kn_readv()* instead.

See Also *kn_connect()*, *kn_readv()*, *kn_recv()*, *kn_socket()*

Purpose **Select Sockets Ready for Receive or Send****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_SOCKET.H*.
`#include "KN_SOCKET.H"`
`int kn_select(int nfds, fd_set *readfds,`
`fd_set *writefds,`
`fd_set *exceptfds,`
`struct timeval *timeout);`

Description *Nfds* indicates the number of sequential socket descriptors which are to be examined in each descriptor set. Since socket descriptors are numbered sequentially from 0, parameter *nfds* must be the numerically largest socket descriptor value, *plus one*. For convenience, you can set *nfds* to *KN_MAX_SOCKET* and KwikNet will examine all sockets identified in each descriptor set.

Readfds is a pointer to a descriptor set which, on entry, identifies the sockets to be interrogated. Upon return, the descriptor set at **readfds* will be updated to identify which of the interrogated sockets have received data ready to be read. A socket will also be declared ready to read if an error other than *EINPROGRESS* or *EWOULDBLOCK* is recorded in the socket while the socket is selected. Set this parameter to *NULL* to ignore sockets which are ready to be read.

Writefds is a pointer to a descriptor set which, on entry, identifies the sockets to be interrogated. Upon return, the descriptor set at **writefds* will be updated to identify which of the interrogated sockets have space available, making the socket ready for sending more data. A socket will also be declared ready for sending if an error other than *EINPROGRESS* is recorded in the socket while the socket is selected. Set this parameter to *NULL* to ignore sockets which are ready for sending.

Exceptfds is a pointer to a descriptor set which, on entry, identifies the sockets to be interrogated. Upon return, the descriptor set at **exceptfds* will be updated to identify which of the interrogated sockets have outstanding exceptions present. KwikNet treats recorded errors other than *EINPROGRESS* and *EWOULDBLOCK* as exceptions. KwikNet also reports an exception if out-of-band data has been received. Set *exceptfds* to *NULL* to ignore sockets with outstanding exceptions.

...more

Description ...continued

Timeout is a pointer to a structure which defines the interval for which the caller is prepared to wait for at least one socket to meet the selected criteria. The timing resolution is determined by the KwikNet clock frequency. Set *timeout* to *NULL* to wait forever. Set the interval value to 0 to return immediately if no sockets are ready. If the specified timeout interval is less than one millisecond, the procedure will return immediately with a timeout indication if no sockets are ready.

Structure *timeval* is described in file *KN_SOCK.H* as follows:

```
struct timeval {
    unsigned long tv_sec;      /* Number of seconds          */
    unsigned long tv_usec;    /* Number of microseconds    */
};
```

Type *fd_set* is defined by a *typedef* in Treck file *TRSOCKET.H*. Variables of type *fd_set* can be manipulated using the following macros which are defined in Treck file *TRSOCKET.H*. In the descriptions which follow, *fdset* is a variable of type *fd_set* and *s* is a valid socket descriptor.

<i>FD_SET(s, &fdset)</i>	Set socket <i>s</i> identifier in variable <i>fdset</i> .
<i>FD_CLR(s, &fdset)</i>	Clear socket <i>s</i> identifier in variable <i>fdset</i> .
<i>FD_ISSET(s, &fdset)</i>	Test socket <i>s</i> identifier in variable <i>fdset</i> .
<i>FD_ZERO(&fdset)</i>	Clear all socket identifiers in variable <i>fdset</i> .

Use *FD_ZERO* to reset (clear) all socket identifiers in a descriptor set. Use *FD_SET* to identify the specific sockets to be interrogated. *FD_ISSET* returns a non-zero value if the identifier for socket *s* is set or zero if the identifier for socket *s* is clear.

The behavior of these macros is undefined if the socket descriptor *s* is invalid.

Note that the structure and parameter names are derived from UNIX for which this procedure interrogates both files and sockets.

...more

Returns If successful, this procedure returns the total number of ready sockets identified in the descriptor sets. The variables **readfds*, **writefds* and **exceptfds* are updated to identify the subset of the sockets specified by the caller which match their respective criteria.

This procedure returns 0 if the specified timeout interval elapses before any sockets are ready.

On failure, the error status -1 is returned. Unfortunately, the error indicator defining the reason for failure cannot be recorded. You cannot use *kn_errno()* to retrieve the error code since you do not have a unique socket descriptor to interrogate. The following error codes, although not available for testing, still define the possible reasons for failure.

<i>EBADF</i>	A socket descriptor in one of the sets is invalid.
<i>ENOBUFS</i>	Memory needed to service the request is unavailable.

See Also *kn_socket()*

Purpose **Send Data to a Connected Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_send(int s, void *buf, int len, int flags);
```

Description *s* is a socket descriptor identifying the connected socket to which data is to be sent.*Buf* is a pointer to the buffer of data to be sent.*Len* is the buffer size, measured in bytes.*Flags* is a control variable used to modify the send process. *Flags* is 0 or the logical OR of any of the following values.

<i>MSG_OOB</i>	Allow sending of out-of-band data.
<i>MSG_DONTWAIT</i>	Send message in non-blocking fashion.

Returns If successful, the number of bytes of data sent from **buf* is returned.
On failure, the error status *-1* is returned.The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	Parameter <i>buf</i> or <i>len</i> is invalid or the buffer length <i>len</i> is declared to be less than 0 or is invalid for the socket's protocol.
<i>ENOTCONN</i>	The socket is not connected.
<i>EHOSTUNREACH</i>	A destination is required but is not available.
<i>EMSGSIZE</i>	The message is larger than the maximum which can be sent atomically as required by the socket protocol.
<i>ESHUTDOWN</i>	The socket has been marked to shut down writing or the socket is in the process of being closed.
<i>EWOULDBLOCK</i>	The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> but there is a need to block the caller to complete the operation.
<i>ENOBUFS</i>	Memory is not available to complete the request.

...more

Restriction If none of the message data can be delivered to the socket, the caller will be blocked unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a *-1* error status and the error code *EWOULDBLOCK* will be stored in the socket descriptor.

See Also `kn_connect()`, `kn_sendto()`, `kn_writev()`, `kn_socket()`

Purpose **Send Data to a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_sendto(int s, void *buf, int len, int flags,
              struct sockaddr *to, int tolen);
```

Description *S* is a socket descriptor identifying the socket to which data is to be sent.
The socket can be connected or unconnected.*Buf* is a pointer to the buffer of data to be sent.*Len* is the buffer size, measured in bytes.*Flags* is a control variable used to modify the send process. *Flags* is 0 or the logical OR of any of the following values.*MSG_OOB* Allow sending of out-of-band data.*MSG_DONTWAIT* Send message in non-blocking fashion.*To* is a pointer to the address of the destination to which the data is to be sent. The format of the IP address in structure *sockaddr* is described in header file *KN_SOCK.H*.This parameter is required to identify the destination address when using a connectionless socket such as that used for UDP datagrams on a socket of type *SOCK_DGRAM*. The format of the address is compatible with that received by procedure *kn_recvfrom()*.*Tolen* is the length of the destination address.

...more

- Returns** If successful, the number of bytes of data sent from **buf* is returned. On failure, the error status *-1* is returned.
- The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.
- | | |
|---------------------|---|
| <i>EBADF</i> | The socket descriptor <i>s</i> is invalid. |
| <i>EINVAL</i> | Parameter <i>buf</i> or <i>to</i> is invalid or the buffer length <i>len</i> is declared to be less than 0 or is invalid for the socket's protocol or parameter <i>to</i> specifies a length that is less than the length of a valid address. |
| <i>EMSGSIZE</i> | The message is larger than the maximum which can be sent atomically as required by the socket protocol. |
| <i>EPROTOTYPE</i> | Request is invalid for sockets using the TCP protocol. |
| <i>EHOSTUNREACH</i> | A destination is required but is not available. |
| <i>EWouldBlock</i> | The socket is marked non-blocking or <i>flags</i> specifies <i>MSG_DONTWAIT</i> but there is a need to block the caller to complete the operation. |
| <i>ENOBUFS</i> | Memory is not available to complete the request. |
- Restriction** If none of the message data can be delivered to the socket, the caller will be blocked unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a *-1* error status and the error code *EWouldBlock* will be stored in the socket descriptor.
- Restriction** This procedure cannot be used to send to a foreign host using the TCP protocol. Use *kn_send()* or *kn_writev()* instead.
- See Also** *kn_connect()*, *kn_send()*, *kn_writev()*, *kn_socket()*

Purpose **Set a Particular Socket Option****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_setsockopt(int s, int level, int optionid,
                  void *optionval, int optionlen);
```

Description *s* is a socket descriptor identifying the socket for which the particular socket option is to be modified.

Level is an identifier indicating the socket or protocol level for which an option must be modified. Use *SOL_SOCKET* for the highest, socket level options. Use *IPPROTO_TCP* for TCP protocol level options. The Turbo Treck TCP/IP Stack also supports the use of *IPPROTO_IP* for IP protocol level options. Other protocol levels are not supported.

Optionid identifies the option which is to be modified. The following option identifiers can be used to set the state of the option or to modify its associated parameter. These options are described in Chapter 5.3. Options marked > are non-standard KADAK extensions. Only one option can be specified in each call. The Turbo Treck TCP/IP Stack supports a number of additional options (at the socket, TCP and IP levels), all of which are described in the Treck TCP/IP User Manual.

<i>SO_BROADCAST</i>	<i>bool</i>	UDP		Permission to broadcast messages
<i>SO_REUSEADDR</i>	<i>bool</i>	UDP	TCP	Local address reuse
<i>SO_KEEPAIVE</i>	<i>bool</i>		TCP	Keep connections alive
<i>SO_OOBINLINE</i>	<i>bool</i>		TCP	Allow out-of-band data in band
<i>SO_LINGER</i>	<i>struct</i>		TCP	Linger on close if data present
<i>SO_SNDBUF</i>	<i>u/s long</i>		TCP	Buffer size for send
<i>SO_RCVBUF</i>	<i>u/s long</i>		TCP	Buffer size for receive
<i>SO_SNDLOWAT</i>	<i>u/s long</i>		TCP	Buffer low limit for send
<i>SO_RCVLOWAT</i>	<i>u/s long</i>		TCP	Buffer low limit for receive
<i>SO_NONBLOCK</i>	<i>bool</i>	UDP	TCP >	Socket is non-blocking mode
<i>TCP_NODELAY</i>	<i>bool</i>		TCP	Do not delay send to coalesce data
<i>TCP_NOPUSH</i>	<i>bool</i>		TCP	Do not push last byte of data sent
<i>TCP_MAXSEG</i>	<i>int</i>		TCP	Set maximum segment size (MSS)

...more

Description ...continued

Optionval is a pointer to the option value. The size of each option is indicated in the option list. Note that *bool* is an *int* value that is *1* if the option is to be enabled or *0* if the option is to be disabled. The type *u/s long* indicates a 32-bit value usually represented by an *unsigned long*. The structure *linger* required by option *SO_LINGER* is described in header file *KN_SOCK.H*.

Optionlen is the length of the option value at the location referenced by *optionval*.

Note Option *SO_NONBLOCK* is a unique KwikNet option which conditions a socket such that subsequent socket operations proceed with or without blocking the caller. Note that, although a socket is always a blocking socket when first created, this option permits the mode of operation to be altered by your application. The option value in the call must be non-zero to set non-blocking mode or zero to restore the socket to blocking mode.

Returns If successful, a value of *0* is returned.
On failure, the error status *-1* is returned.

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	One or more parameters are invalid or parameter <i>optionlen</i> specifies a length that is less than that required to specify the option.
<i>ENOPROTOOPT</i>	The option is unknown at the level indicated.
<i>EPERM</i>	The option cannot be set after a connection is established.

See Also *kn_getsockopt()*, *kn_socket()*

Purpose	Shut down All or Part of a Full Duplex Socket Connection								
Used by	<div> <div>■ Task</div> <div>□ ISP</div> <div>□ Timer Procedure</div> <div>□ Restart Procedure</div> <div>□ Exit Procedure</div> </div>								
Setup	Prototype is in file <i>KN SOCK.H</i> . <pre>#include "KN SOCK.H" int kn_shutdown(int s, int how);</pre>								
Description	<p><i>s</i> is a socket descriptor identifying the socket with the connection which is to be shut down.</p> <p><i>How</i> is an integer which defines how the connection is to be adjusted.</p> <ul style="list-style-type: none"> 0 if no further receives are allowed 1 if no further sends are allowed 2 if no further receives or sends are allowed 								
Returns	<p>If successful, a value of 0 is returned.</p> <p>On failure, the error status -1 is returned.</p> <p>The error indicator for socket <i>s</i> is set to define the reason for failure. Use <i>kn_errno()</i> to retrieve the error code.</p> <table> <tr> <td><i>EBADF</i></td> <td>The socket descriptor <i>s</i> is invalid.</td> </tr> <tr> <td><i>EINVAL</i></td> <td>Parameter <i>how</i> is invalid.</td> </tr> <tr> <td><i>ESHUTDOWN</i></td> <td>The connection has been shut down or the socket is in the process of closing.</td> </tr> <tr> <td><i>EOPNOTSUPP</i></td> <td>Invalid operation (not a TCP socket).</td> </tr> </table>	<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.	<i>EINVAL</i>	Parameter <i>how</i> is invalid.	<i>ESHUTDOWN</i>	The connection has been shut down or the socket is in the process of closing.	<i>EOPNOTSUPP</i>	Invalid operation (not a TCP socket).
<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.								
<i>EINVAL</i>	Parameter <i>how</i> is invalid.								
<i>ESHUTDOWN</i>	The connection has been shut down or the socket is in the process of closing.								
<i>EOPNOTSUPP</i>	Invalid operation (not a TCP socket).								
Restriction	This procedure can only be used to shut down a socket which supports the TCP protocol.								
See Also	<i>kn_accept()</i> , <i>kn_connect()</i> , <i>kn_socket()</i>								

Purpose **Create a Socket (an Endpoint for Communication)****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_socket(int domain, int type, int protocol);
```

Description *Domain* specifies the communications domain within which communication will occur. The domain identifies the protocol family which should be used. The protocol family generally matches the address family for the addresses supplied in subsequent socket operations. The only protocol family support by KwikNet is the ARPA Internet Protocol, identified as *PF_INET* . The corresponding address family is *AF_INET*.

Type defines the semantics of communication supported by the socket. *Type* must be *SOCK_STREAM* for use with the TCP protocol or *SOCK_DGRAM* for use with the UDP protocol. The Turbo Treck TCP/IP Stack also supports raw sockets which are identified as type *SOCK_RAW* and used with the IP protocol.

The *SOCK_STREAM* type of socket provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism can be supported.

The *SOCK_DGRAM* type of socket supports datagrams: connectionless, unreliable messages of a fixed (typically small) maximum length.

The *SOCK_RAW* type of socket supports datagram at the IP protocol layer, giving you full access to low level Internet services.

Protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified. Pick the protocol from the following table.

Family	Type	Protocol	Used for
<i>PF_INET</i>	<i>SOCK_DGRAM</i>	<i>IPPROTO_UDP</i>	UDP over IP
<i>PF_INET</i>	<i>SOCK_STREAM</i>	<i>IPPROTO_TCP</i>	TCP over IP
<i>PF_INET</i>	<i>SOCK_RAW</i>	<i>IPPROTO_ICMP</i>	ICMP over IP
<i>PF_INET</i>	<i>SOCK_RAW</i>	<i>IPPROTO_IGMP</i>	IGMP over IP

...more

Returns If successful, a non-negative socket descriptor is returned.
On failure, the error status `-1` is returned.

If a socket cannot be created, the error indicator defining the reason for failure cannot be recorded. You cannot use `kn_errno()` to retrieve the error code since you have no socket descriptor to interrogate. The following error codes, although not available for testing, still define the possible reasons for failure.

<code>EPROTONOSUPPORT</code>	The socket <i>type</i> or the specified <i>protocol</i> is not supported within the <i>domain</i> .
<code>EMFILE</code>	No more sockets are available. Close some sockets or revise your KwikNet configuration to provide more.
<code>ENOBUFFS</code>	Resources needed to create a socket are unavailable.

Restriction Sockets of type `SOCK_RAW` cannot be used unless you have configured your KwikNet libraries to allow support raw sockets. Edit your KwikNet Network Parameter File and check the box labeled "Use raw sockets" on the TCP property page.

See Also `kn_accept()`, `kn_bind()`, `kn_connect()`

Purpose **Send Scattered Data to a Socket****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure**Setup** Prototype is in file *KN_SOCK.H*.

```
#include "KN_SOCK.H"
int kn_writev(int s, struct iovec *iovecp, int iovcnt);
```

Description *s* is a socket descriptor identifying the socket to which data is to be sent.
The socket must be connected.

iovecp is a pointer to an array of data vectors describing the locations of the data to be sent. Each data vector is an *iovec* structure which is described in file *KN_SOCK.H* as follows:

```
struct iovec {
    void    *iov_base;           /* Data pointer          */
    int     iov_len;            /* Length of data        */
};
```

iov_base is a pointer to data to be sent.

iov_len is the data buffer size, measured in bytes.

iovcnt is an integer defining the number of data vectors (*iovec* structures) which are provided in the array referenced by parameter *iovecp*.

Returns If successful, the total number of bytes of data sent is returned.
The array of *iovec* structures at **iovecp* is unaltered.

Upon successful return, you must examine the data vectors to determine if the total number of bytes in the array of vectors matches the number just sent. If not, you must identify the *iovec* structure at which the current operation ceased. You must then adjust the data pointer and length of that vector to bypass the fraction of the data which has been sent. You can then make another request to send the remainder, starting at the corrected position in that vector.

On failure, the error status *-1* is returned.

...more

Returns ...continued

The error indicator for socket *s* is set to define the reason for failure. Use *kn_errno()* to retrieve the error code.

<i>EBADF</i>	The socket descriptor <i>s</i> is invalid.
<i>EINVAL</i>	Parameter <i>iovecp</i> or <i>iovcnt</i> is invalid or the buffer length in a data vector was declared to be less than 0 or the sum of the data vector lengths exceeds the maximum size of an integer.
<i>EMSGSIZE</i>	The message is larger than the maximum which can be sent atomically as required by the socket protocol.
<i>ENOTCONN</i>	The socket is not connected.
<i>EWOULDBLOCK</i>	The socket is marked non-blocking but there is a need to block the caller to complete the operation.
<i>ESHUTDOWN</i>	The socket has been marked to shut down writing or the socket is in the process of being closed.
<i>ENOBUFS</i>	Memory is not available to complete the request.

Restriction If none of the message data can be delivered to the socket, the caller will be blocked unless the socket *s* is marked as non-blocking. In the latter case, the caller will resume with a -1 error status and the error code *EWOULDBLOCK* will be stored in the socket descriptor.

Restriction On 16-bit processors, the amount of data which can be sent is restricted to 32767 bytes because the value returned by *kn_writev()* is a signed integer.

See Also *kn_connect()*, *kn_readv()*, *kn_send()*, *kn_sendto()*, *kn_socket()*

6. KwikNet PPP Option

6.1 Introduction to PPP

The Point-to-Point Protocol (PPP) is a network protocol used to control the delivery of IP datagrams between two host computers interconnected by a serial link. The KwikNet PPP option adds this protocol to the KwikNet TCP/IP Stack, a compact, reliable, high performance TCP/IP stack, well suited for use in embedded networking applications. PPP offers improved reliability and security features not found in SLIP, the Serial Line Internet Protocol included with KwikNet.

The KwikNet PPP option is best used with a real-time operating system (RTOS) such as KADAK's AMX™ Real-Time Multitasking Kernel. However, the KwikNet PPP option can also be used in a single threaded environment without an RTOS. The KwikNet Porting Kit User's Guide describes the use of KwikNet with your choice of RT/OS. Note that throughout this chapter, the term RT/OS is used to refer to any operating system, be it a multitasking RTOS or a single threaded OS.

You can readily tailor the KwikNet stack to accommodate your PPP needs by using the KwikNet Configuration Builder, a Windows® utility which makes configuring KwikNet a snap. Your KwikNet stack will only include the PPP features required by your application.

The Point-to-Point Protocol (PPP) is formally defined by the IETF document RFC-1661. That document describes the Link Control Protocol (LCP) used by PPP to negotiate the configuration of the link over which communication will occur.

IETF document RFC-1334 describes the Password Authentication Protocol (PAP) first used by PPP to provide link authentication prior to network use. That protocol has been updated by the Challenge-Handshake Authentication Protocol (CHAP) described by IETF document RFC-1994.

The MS-CHAP variation of the CHAP protocol introduced by Microsoft Corporation and described in IETF document RFC-2433 is also supported by the KwikNet PPP option.

The Extensible Authentication Protocol (EAP), described in IETF document RFC-3748, defines the manner in which a uniquely specified authentication sequence can be used by two PPP peers, as long as each is cognizant of the extended specification.

The IP Control Protocol (IPCP), described in IETF document RFC-1332, defines the manner in which IP datagrams are delivered over the link established by PPP. KwikNet also supports the Name Server Address extension to IPCP as defined in RFC-1877.

The KwikNet PPP option is compliant with these specifications. The RFCs should be consulted for any detailed questions concerning the PPP protocol.

Beyond the summary above, this manual makes no attempt to describe the Point-to-Point Protocol (PPP), what it is or how it operates. It is assumed that you have a working knowledge of the PPP protocol as it applies to your needs. Reference materials are provided in Appendix A.

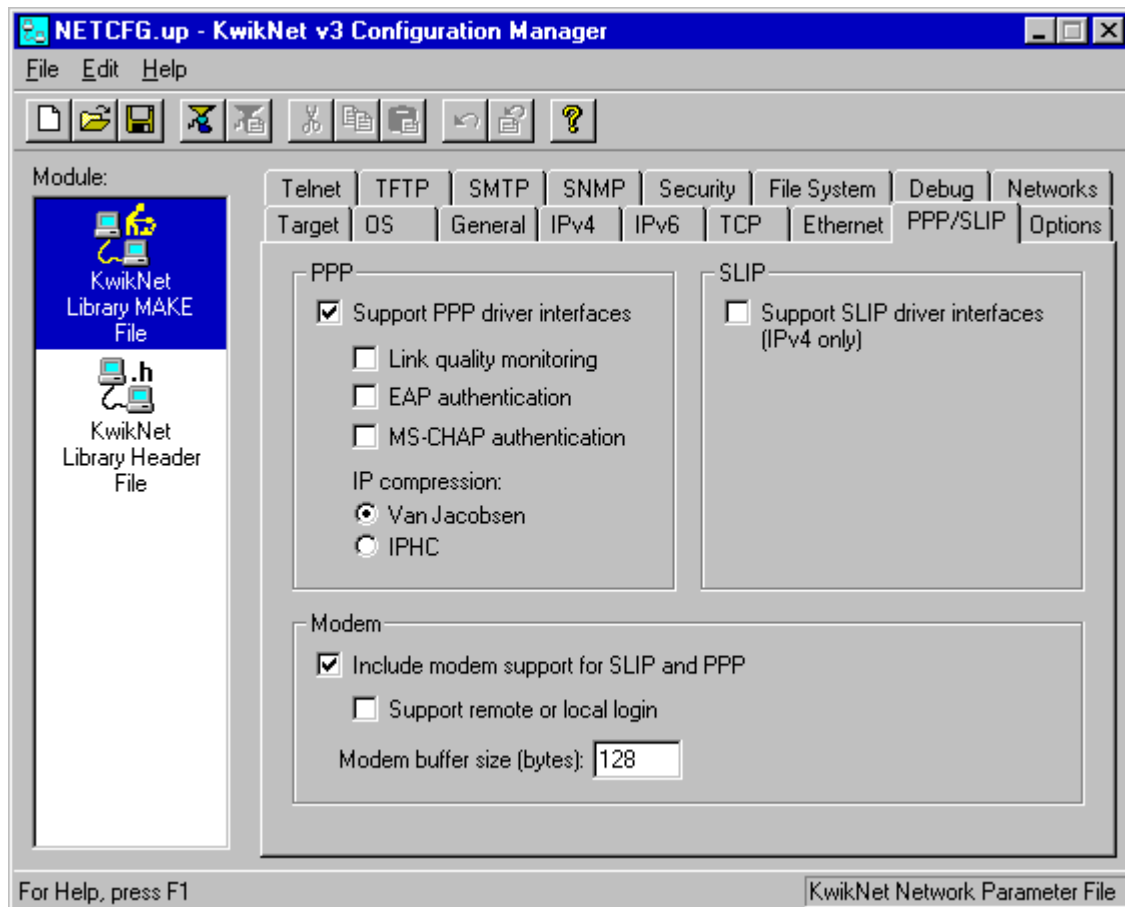
The purpose of this manual is to provide the system designer and applications programmer with the information required to properly configure and implement a networking system using the KwikNet TCP/IP Stack and PPP. It is assumed that you are familiar with the architecture of the target processor.

KwikNet and its options are available in C source format to ensure that regardless of your development environment, your ability to use and support KwikNet is uninhibited.

The C programming language, commonly used in real-time systems, is used throughout this manual to illustrate the features of KwikNet and its PPP option.

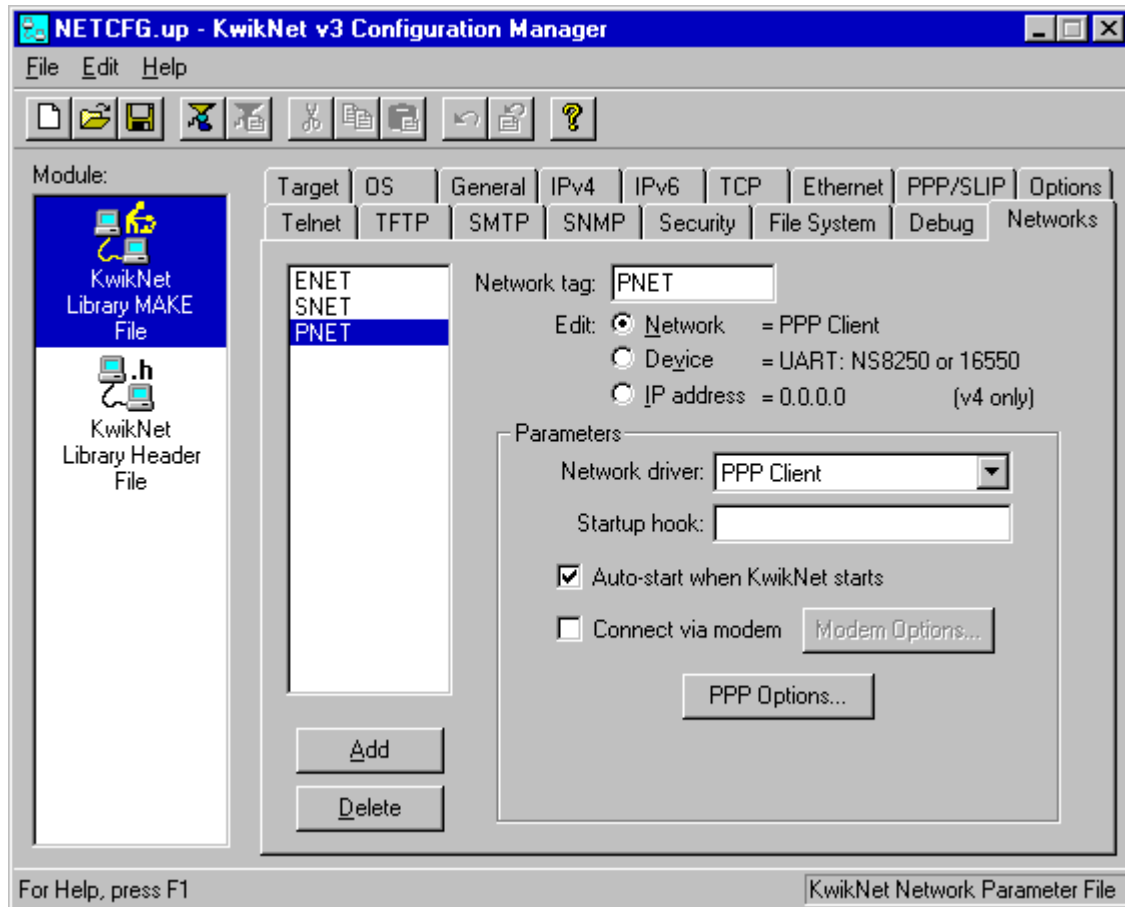
6.2 KwikNet PPP Configuration

You can readily tailor the KwikNet stack to accommodate your PPP needs by using the KwikNet Configuration Builder to edit your KwikNet Network Parameter File. The KwikNet Library parameters are edited on the PPP/SLIP property page. The PPP parameters have been defined in Chapter 2.3. The layout of the window is repeated below for convenience.



PPP Network Definition

You must define each PPP network which your application must support. A separate definition is required for each such network. The total number of networks must not exceed the maximum number of networks which your KwikNet Network Parameter File allows. A PPP network can be defined using the Networks property page as described in Chapter 2.6. The layout of the window is repeated below for convenience.



PPP Options

For each KwikNet network which supports PPP, you must define the PPP parameters which govern its use. These PPP option parameters can be edited using the PPP Dialog Box entered via the PPP Options... button on the Networks definition property page. These PPP options have been described in Chapter 2.6. The layout of the dialog box is repeated below for convenience.

PPP Network Options

LCP Negotiation

- ☒ Header field compression
- ☒ Magic number negotiation (turn off for loopback)

Authentication

Require		Accept
<input type="checkbox"/> <---	EAP	---> <input type="checkbox"/>
<input type="checkbox"/> <---	MS-CHAP v1	---> <input type="checkbox"/>
<input type="checkbox"/> <---	CHAP	---> <input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> <---	PAP	---> <input checked="" type="checkbox"/>

Require = Peer must authenticate itself using one of these protocols.
Accept = Local network will authenticate using one of these protocols.

IPCP Negotiation

- ☒ IP compression
- ☒ Request DNS server information from peer
- ☒ Set peer as default gateway

Local IP address: 0 . 0 . 0 . 0

Remote IP address: 0 . 0 . 0 . 0

Primary DNS server: 0 . 0 . 0 . 0

Secondary DNS server: 0 . 0 . 0 . 0

OK Cancel

6.3 Using a PPP Network Interface

The PPP Client and Server

The easiest way to add a PPP network interface to your KwikNet application is to use the KwikNet Configuration Manager to specify the network in your Network Parameter File. Each network interface defined in this manner is automatically prebuilt for you whenever KwikNet starts. However, your application can dynamically add a PPP network interface at runtime by calling KwikNet procedure `kn_ifadd()`.

No matter how the network interface is added, it does not become operational until it is opened. KwikNet automatically opens each network interface which it prebuilds if, and only if, the network definition indicates that the network must auto-start. Network interfaces that are dynamically added cannot be auto-started.

A PPP network interface operates in a client-server relationship with its peer. The network interface which initiates a PPP connection is considered to be the client. The peer which accepts the client's request is considered to be the server.

The PPP client-server relationship is quite obvious when a dialing modem is used with the serial device driver attached to the network interface.

The network interface is a **PPP client** if it has to dial out to get the attention of its peer.

The network interface is a **PPP server** if it has to auto answer a call from its peer.

Separate network interfaces are required for each PPP client and server. Therefore, if you must initiate a PPP connection you will require a network interface configured as a PPP client. If you must also accept a request for a PPP connection, you will require a separate network interface configured as a PPP server. Fortunately, most applications require only a PPP client or a PPP server, not both on a single serial line.

Warning

If two PPP network interfaces share a common serial interface, only one of the network interfaces must be open at any instant. Managing the opening and closing of the network interfaces to switch between the PPP client and server is the responsibility of your application.

Note

SLIP network interfaces are used in a manner similar to that described for PPP network interfaces. However, a single SLIP network interface can be used as either a SLIP client or as a SLIP server. SLIP networks do not support any form of peer authentication.

Opening a PPP Network Interface

The KwikNet PPP network driver supervises the orderly opening of the PPP network interface. During the opening process, your application is given several opportunities to adjust the manner in which the PPP connection is eventually established.

The network driver calls the **network startup hook**, if one has been configured for use by the network interface. The startup hook is specified in the network definition for prebuilt PPP networks. If the network interface is dynamically added at runtime, a network startup hook can be registered by calling KwikNet procedure *kn_ifnethook()*.

The network startup hook is described in Appendix A.1 of the KwikNet Device Driver Technical Reference Manual. The parameters which it receives have been documented in Chapter 4.5 of this manual.

In the absence of an application specific network hook, the network driver calls its own default hook, *kn_ppp_default_hook()*. This KwikNet procedure registers the PPP negotiation options which you have specified that the PPP stack must follow to establish the PPP connection. This hook then returns to the network driver leaving the configured network parameters and options unaltered.

Your network hook can alter the network parameters and options which it receives. It is the responsibility of the hook to register the required PPP negotiation options with the PPP protocol stack. You can do so by calling the default KwikNet hook procedure *kn_ppp_default_hook()*, passing it the network parameters which your hook has received, whether modified or not. Of course, your hook function can register the PPP options itself with the appropriate calls to the Treck PPP service procedures described in Chapter 7 of the Treck TCP/IP User Manual. You should review the source code of function *kn_ppp_default_hook()* in KwikNet file *PPP\KN_PPPAA.C* for an excellent example of the use of the Treck API.

Once the network parameters and PPP negotiation options have been established, the network driver opens the attached serial device driver to prepare the underlying physical device interface for use. The device driver will call your **device startup hook**, if one has been provided, giving you an opportunity to modify the device driver's operating characteristics. The device startup hook is described in Appendix A.2 of the KwikNet Device Driver Technical Reference Manual.

The network driver then opens the modem driver if it has been attached to the network interface. The modem driver will call your **modem startup hook**, if one has been provided, giving you an opportunity to modify the modem driver's dialing or auto answering specifications. The modem driver will then dial out to connect to the peer or enter the auto answer state to await a dial-in request from the peer. The modem startup hook is described in Appendix A.3 of the KwikNet Device Driver Technical Reference Manual.

Once a communication connection with the peer is in place, the network driver allows the PPP protocol stack to begin the negotiation of a PPP connection with the peer. As the negotiation progresses, KwikNet may have to authenticate itself for its peer. KwikNet may also be required to authenticate the peer. In either case, KwikNet will call an administrative procedure *kn_ppp_admin()* for assistance in the authentication process. This procedure is described in Chapter 6.4.

6.4 PPP Authentication Parameters

If any of your PPP networks require PAP, CHAP, MS-CHAP or EAP authentication, you must provide the KwikNet PPP network driver with access to the authentication parameters needed to support these protocols. These parameters are the heart of your PPP security system. The maintenance and safekeeping of these parameters is your responsibility.

The KwikNet PPP authentication parameters for each PPP network interface are located in array *kn_ppp_auth[]* in module *KN_PPPAA.C* located in the KwikNet PPP installation directory. The authentication parameters are specified in a *knx_ppp_admin* structure which is defined in module *KN_PPPAA.C*.

```
struct knx_ppp_admin {                /* PPP administration structure */
    const char xad_tag[KN_TAGSIZE];    /* PPP network tag                */
    struct knx_authnset xad_pap;        /* PAP parameters                  */
    struct knx_authnset xad_chap;       /* CHAP parameters                 */
#ifdef TM_USE_PPP_MSCHAP
    struct knx_authnset xad_mschap1;    /* MS-CHAP v1 parameters          */
#endif
#ifdef TM_USE_EAP
    struct knx_authnset xad_eapmd5;     /* EAP-MD5 parameters             */
#endif
};
```

The network tag is provided to identify the PPP network interface to which the authentication parameters apply. This tag must exactly match the tag used in the network definition when the network interface was added to your KwikNet configuration.

For each supported authentication protocol, you must provide the authentication parameters in a *knx_authnset* structure which is defined in module *KN_PPPAA.C*. The structure provides the name and password (secret) for both the local network interface and the peer. Some protocols also require the challenge name.

```
struct knx_authnset {                 /* PPP authentication set          */
    struct knx_authn *xad_lname;       /* Local name                      */
    struct knx_authn *xad_lsecret;     /* Local secret                    */
    struct knx_authn *xad_pname;       /* Peer name                       */
    struct knx_authn *xad_psecret;     /* Peer secret                     */
    struct knx_authn *xad_phost;       /* Challenge name                  */
};
```

Each authentication parameter is specified as a constant array of characters. The parameter is provided in a *knx_authn* structure which is defined in module *KN_PPPAA.C*. The structure provides the length of the authentication parameter and a pointer to the character array which constitutes the parameter.

```
struct knx_authn {                    /* PPP authentication parameter    */
    short int xad_length;              /* Length of parameter             */
    short int xad_rsv1;                /* Reserved for alignment          */
    const char *xad_value;             /* Pointer to parameter value      */
};
```

File *KN_PPPAA.C* provides the authentication parameters for one user on a single PPP network interface. You must edit this module to define the authentication parameters which apply to each of your PPP network interfaces.

The KwikNet PPP network driver accesses these authentication parameters by calling the KwikNet PPP administration function *kn_ppp_admin()* within module *KN_PPPAA.C*. You must review this function and, if necessary, alter it to meet your specific requirements. A description of this function is presented later in this chapter.

Security Issues

Since the PPP administration function *kn_ppp_admin()* is the funnel through which all PPP authentication parameters flow, you may wish to enhance this function to include security features to prevent unauthorized access to this critical information. For example, your authentication parameters might be derived by reading the magnetic strip on an authorization card.

If your network supports both local and peer authentication, you should use different authentication parameters for the local network and its peer. If one set of parameters is used to authenticate both, then the security provided by PPP is trivial to crack.

If your network supports multiple authentication protocols, you should use different authentication parameters for each to avoid a security breach. Suppose that you use the same passwords for both PAP and CHAP. Since PAP sends the password as unencrypted text, a "rogue peer" could negotiate PAP to obtain the password and then use the password as the secret for subsequent CHAP negotiations with a more secure authenticator.

For more information on PPP security issues, refer to pages 66-68 in Carlson's book *PPP Design and Debugging* (see Appendix A of the KwikNet TCP/IP Stack User's Guide).

Purpose **Provide PPP Authentication Services**

Context ■ KwikNet Task ■ Application Task

Setup Prototype and definitions are in file *KN_API.H*.
 Structure definitions are in file *KN_PPPAA.C*.

```
#include "KN_LIB.H"
#include "KN_NET.H"
int kn_ppp_admin(KN_NETDP netdp, int opcode,
                 const char *instr, int inlen,
                 const char **outbuf, int buflen);
```

Description *Netdp* is a network descriptor pointer, the KwikNet handle used to identify the network interface of interest.

Opcode is one of the following operation codes:

<i>KN_PPPAUTH_LPNAME</i>	PAP: get local name
<i>KN_PPPAUTH_LPPWD</i>	PAP: get local password
<i>KN_PPPAUTH_PPCHECK</i>	PAP: authenticate peer
<i>KN_PPPAUTH_LCNAME</i>	CHAP or MS-CHAPv1: get local name
<i>KN_PPPAUTH_LCSECRET</i>	CHAP: get local secret
<i>KN_PPPAUTH_PCNAME</i>	CHAP: get challenge name
<i>KN_PPPAUTH_PCSECRET</i>	CHAP: get peer secret
<i>KN_PPPAUTH_LECNAME</i>	EAP-MD5: get local name
<i>KN_PPPAUTH_LECSECRET</i>	EAP-MD5: get local secret
<i>KN_PPPAUTH_PECNAME</i>	EAP-MD5: get challenge name
<i>KN_PPPAUTH_PECSECRET</i>	EAP-MD5: get peer secret
<i>KN_PPPAUTH_LM1SECRET</i>	MS-CHAPv1: get local secret
<i>KN_PPPAUTH_PM1SECRET</i>	MS-CHAPv1: get peer secret

Instr is a pointer to a character array needed to process the request specified by *opcode*. If no such parameter is required, *instr* will be *NULL*.

Inlen is the length of the character array, if any, referenced by parameter *instr*.

Outbuf is a pointer to a character pointer used to process the request specified by *opcode*. For most operations, *outbuf* is a pointer to storage for a pointer to the authentication parameter being returned to the caller. If *opcode* is *KN_PPPAUTH_PPCHECK*, *outbuf* is a pointer to an additional input parameter.

Buflen is the length of the character array referenced by parameter *outbuf* if, and only if, *outbuf* is used as an input parameter.

...more

Description ...continued

Interpretation of parameters *instr* and *outbuf* for each value of *opcode* is as follows.

For operation codes *KN_PPPAUTH_LPNAME*, *KN_PPPAUTH_LCNAME* and *KN_PPPAUTH_LECNAME*, *instr* and *buflen* are unused. A pointer to the local name for the specified authentication protocol is stored at **outbuf*. Note that CHAP and MS-CHAP share a common local name. Hence, if MS-CHAP is used but CHAP is not used, this function must provide the CHAP local name when requested. The name is returned for delivery to the peer.

For operation code *KN_PPPAUTH_LPPWD*, *KN_PPPAUTH_LCSECRET*, *KN_PPPAUTH_LECSECRET* and *KN_PPPAUTH_LM1SECRET*, *instr* and *buflen* are unused. A pointer to the local password or secret for the specified authentication protocol is stored at **outbuf*. The PAP password is returned for delivery to the peer. For the other protocols, the secret will be used to encrypt the response to the peer's challenge.

For operation code *KN_PPPAUTH_PCSECRET*, *KN_PPPAUTH_PECSECRET* and *KN_PPPAUTH_PM1SECRET*, *instr* is a pointer to the peer name which is of length specified by *inlen*. Parameter *buflen* is unused. If the peer's name matches the peer name for the specified authentication protocol, a pointer to the peer's secret is stored at **outbuf*. It will be used to encrypt the challenge which was sent to the peer. The result is then compared to the challenge response from the peer.

! Important note: see restriction on next page !

For operation code *KN_PPPAUTH_PPCHECK*, *instr* is a pointer to the PAP name which is of length specified by *inlen*. Parameter *outbuf* is a pointer to the PAP password provided by the peer for authentication. The password is of length specified by *buflen*. Note that in this special case, *outbuf* is a pointer to a character array; it is **not a pointer to a pointer**. These parameters must be checked for validity and the result of the check must be returned to the caller. The input parameters at **instr* and **outbuf* must not be modified.

! Important note: see restriction on next page !

For operation code *KN_PPPAUTH_PCNAME* and *KN_PPPAUTH_PECNAME*, *instr* and *buflen* are unused. A pointer to the challenge name for the specified authentication protocol is stored at **outbuf*. Note that CHAP and MS-CHAP share a common challenge name. Hence, if MS-CHAP is used but CHAP is not used, this function must provide the CHAP challenge name when requested. The challenge name is returned for delivery to the peer in the authentication challenge.

...more

...continued

Returns For operations which return a pointer to an authentication parameter in the storage at **outbuf*, an integer *n* is returned. The value of *n* reflects the success or failure of the request.

<i>n</i> >= 0	Length of the authentication parameter referenced by the character pointer stored at <i>*outbuf</i>
<i>n</i> = -1	Authentication parameter is not available

For operation code *KN_PPPAUTH_PPCHECK*, an integer *n* is returned which reflects the success or failure of the validity check requested.

<i>n</i> = 1	The peer's PAP name and password are valid.
<i>n</i> = 0	The peer's PAP name and password are NOT valid.

Restriction When called upon to authenticate the peer's name, this function does not receive a network descriptor pointer. Parameter *netdp* is *NULL*. The sample implementation of procedure *kn_ppp_admin* in file *KN_PPPAA.C* unconditionally compares the peer name with that specified for the first network interface described in array *kn_ppp_auth[]*.

6.5 Adding PPP to Your Application

Before you add the PPP protocol to your application, you must have a working KwikNet IP stack (with or without TCP) operating with your RT/OS and your serial device driver. It is imperative that you start with a tested TCP/IP stack with functioning device drivers before you add PPP. If these components are not operational, the KwikNet PPP option cannot operate correctly.

KwikNet Library

Begin by deciding which PPP features must be supported. Review the PPP property page described in Chapter 2.3 and the PPP Options... dialog described in Chapter 2.6. In particular, omit support for link quality monitoring and EAP and MS-CHAP authentication unless you actually have such a need. The memory savings are significant.

If any of your PPP network interfaces must support PAP, CHAP, MS-CHAP or EAP authentication, edit file *KN_PPPAA.C* in KwikNet directory *PPP* as described in Chapter 6.4 to provide access to your administration function *kn_ppp_admin()*.

Armed with your PPP feature list, use the KwikNet Configuration Manager to edit your application's KwikNet Network Parameter File to include the PPP protocol. If desired, define all PPP network interfaces to be prebuilt by KwikNet when it starts. Then rebuild your KwikNet Library. The library extension may be *.A* or *.LIB* or some other extension dictated by the toolset which you are using.

Reconstructing Your KwikNet Application

If you have not already done so, add your network serial device drivers to your application. Start by reviewing your KwikNet board driver module *KN_BOARD.C*. Unless you have modified the board driver, it supports four network interrupt sources. If necessary, increase the number of supported network interrupt sources to allow support for your network serial device drivers. If you are porting KwikNet for use with your choice of RT/OS, your KwikNet OS Interface Module *KN_OSIF.C* must be updated to support these additional network interrupt sources. If changes are made, be sure to rebuild your KwikNet Library and compile board driver module *KN_BOARD.C*.

Your application link and/or locate specification file must include the KwikNet Library which you built with support for PPP. The object modules for your network serial device drivers must also be included in your link specification together with your other application object modules.

With these changes in place, you can link and create an updated KwikNet application with PPP support included.

AMX Considerations

When reconstructing a KwikNet application which uses the AMX Real-Time Multitasking Kernel, adapt the procedure just described to include the following considerations.

You may have to edit your AMX User Parameter File to increase the size of the AMX Interrupt Stack and the KwikNet Task stack to meet the needs of the PPP protocol. You must then rebuild and compile your AMX System Configuration Module.

No changes to your AMX Target Configuration Module are required to support PPP provided that your network serial device drivers are already part of your application.

If the KwikNet board driver module *KN_BOARD.C* is altered to increase the number of supported network interrupt sources, your AMX Target Parameter File must be edited to add the corresponding ISP definitions. If changes are made, be sure to rebuild and compile your AMX Target Configuration Module and compile board driver module *KN_BOARD.C*. When using KwikNet with AMX, there is no need to edit the KwikNet OS Interface Module *KN_OSIF.C*.

7. KwikNet Virtual File System

7.1 Introduction

For embedded systems which may not need a full-featured file system, KwikNet offers a Virtual File System (VFS) which can provide access to a limited set of read-only files built into the application.

A virtual file is a duplicate of a real disk file which you have on your development system. However, the virtual file does not reside on a disk. Instead, it sits in the memory of your target system.

Virtual files are created using the KwikNet VFS Generator, a utility program which executes on your development system. You create a simple text file which defines the set of real disk files which are to make up your Virtual File System. The VFS Generator reads this description and produces a C source file which, when compiled and linked with your application, forms your Virtual File System.

You can also use the VFS Generator to compress text files thereby reducing the memory required for your VFS image. Such compression is especially suitable for HTML files which contain frequently repeated HTML tags.

The Virtual File System can be used with or without a real file system. It will forward file requests which it is not equipped to handle, through the KwikNet Universal File System Interface to the underlying file system, if one exists. For example, if you are using the VFS with the AMX/FS File System, a file will be fetched via AMX/FS if it is not first found within the Virtual File System.

VFS File System Structure

The Virtual File System consists of a set of files organized into one or more volumes. In most applications, all files are contained in a single volume. If you wish to support multiple volumes, refer to Chapter 7.4.

The Virtual File System provides a hierarchical directory structure to allow virtual files to be organized into directories as in a conventional file system. However, it does so without actually maintaining directories.

The Virtual File System does not include any services for manipulating directories. For example, you cannot create or delete directories. Furthermore, the VFS has no concept of a current working directory.

The Virtual File System does not support the concept of a disk drive such as the conventional MS-DOS[®] disk drive identified by a drive letter such as *c:*. Hence there is no such concept as the current drive. However, the VFS file naming rules do allow files to be grouped as though drives are supported.

VFS File Names

The file name of a virtual file is a text string consisting of two parts, the volume base and the file part, separated by the / or \ character.

The **volume base** is a string which will only be used by the Virtual File System when your application is running. Its main purpose is to allow you to distinguish your virtual files from real files. If you are not using a real file system, the volume base can be omitted. The volume base is common to all files in a VFS volume.

The **file part** is used to uniquely identify each file. The file part always mimics the path, file name and extension used to identify real files. The reason is as follows. When defining the set of files which a VFS volume will contain, you must use the file naming rules dictated by the system on which you are running the VFS Generator so that the generator will be able to access your real files. Consequently, the file part of the virtual file will always be a portion of the path and file name used to identify the real file from which the virtual file was derived.

The following examples illustrate valid virtual file names and their parts.

Volume Base	File Part	File Name
<i>\vfsroot</i>	<i>index.htm</i>	<i>\vfsroot\index.htm</i>
<i>C:\vfsroot</i>	<i>html\KADAK.HTM</i>	<i>C:\vfsroot\html\KADAK.HTM</i>
<i>\</i>	<i>index.htm</i>	<i>\index.htm</i>
	<i>image/graphic.gif</i>	<i>/image/graphic.gif</i>

By default, the **VFS separator** character is the separating slash of the underlying real file system, if any. For example, if MS-DOS® or the AMX/FS File System is used with the Virtual File System, then the VFS separator will be the \ character. If no real file system is present, the VFS separator will be the / character.

File names within the Virtual File System are adjusted to **local form** as follows. The volume base will have a trailing slash added to it if none was present in its definition. All forward and backward slash characters (/ or \) in the volume base and in all file parts will be converted to the VFS separator character. The file name is created by appending the file part to the volume base. No other character translation is done.

File names are maintained in local form so that references to files which are not present in the Virtual File System can always be handled by the underlying real file system, if one exists.

The case of alphabetic characters within a virtual file's name is not altered by the Virtual File System. However, case is ignored when searching for a virtual file. Hence, virtual file names appear to the application to be case insensitive. But beware! If the Virtual File System cannot find a particular file, the request will be passed to the underlying file system which may very well be case sensitive.

VFS File Operations

The Virtual File System provides a set of file access functions with names of the form *kvf_XXXXX()*. These functions, listed in Figure 7.1-1, are described in detail in Chapter 7.5. Function prototypes are located in header file *KN_FILES.H* in the KwikNet *TCPIP* installation directory.

Your application can use these functions to access virtual files or real files. If the referenced file is not a virtual file, the request will be passed on to the equivalent function in the underlying file system, if it exists. Otherwise, an error response will be produced.

Your application can use these functions even if your KwikNet Library has been configured to exclude the Virtual File System. In such cases, the functions map to their equivalent functions in the underlying file system, if one exists.

VFS Function	Purpose
<i>kvf_close</i>	* Close a virtual file
<i>kvf_fsize</i>	* Fetch the size of a virtual file
<i>kvf_open</i>	* Open a virtual file
<i>kvf_read</i>	* Read from a virtual file
<i>kvf_seek</i>	* Seek within a virtual file
<i>kvf_tell</i>	* Determine the current position of the file pointer in a virtual file
<i>kvf_flush</i>	Flush a file
<i>kvf_remove</i>	Remove (delete) a file
<i>kvf_rename</i>	Rename a file
<i>kvf_write</i>	Write to a file
<i>kvf_isdir</i>	Check if filename is a directory
<i>kvf_mkdir</i>	Make a directory
<i>kvf_rmdir</i>	Remove (delete) a directory
<i>kvf_opendir</i>	Open a directory for listing
<i>kvf_closedir</i>	Close a directory when finished listing
<i>kvf_readdir</i>	Read next directory listing entry
<i>kvf_isvfile</i>	*! Determine if a file is a virtual file
<i>kvf_voladd</i>	*! Add a VFS volume
<i>kvf_voldel</i>	*! Delete a VFS volume
Note	* Supported by Virtual File System ! Not supported by Universal File System

Figure 7.1-1 KwikNet Virtual File System Functions

VFS File Access Rights

User access to files within the KwikNet Virtual File System is governed by the KwikNet Administration Interface as described in Appendix D. The access rules are the same for all file systems.

Each user is identified with a user name and password and is given specific access rights. Each user can also be assigned a base directory with the user's access restricted to only those files contained within that base directory and its subdirectories.

The definitions of your KwikNet users and their file access rights can have a significant impact on your Virtual File System. For example, if you declare that a user has base directory *E:\userbase* but create a Virtual File System with a volume base of *\vfsroot*, that user will never be able to access any virtual files. The reason is simple. All virtual files will have names like *\vfsroot\anyfile* but the user can only access files with names of the form *E:\userbase\anyfile*.

If a user must be allowed access to both virtual files and real files, then the Virtual File System volume base must, at the very least, match all or part of the user's base directory. For example, if you declare that a user has base directory *E:\userbase* and create a Virtual File System with a volume base of *E:*, then that user will be able to access all virtual or real files of the form *E:\userbase\anyfile*.

7.2 Virtual File System Definition

Each KwikNet Virtual File System volume is defined by directives contained in a text file called a VFS Definition File (VDF). A typical VFS Definition File is illustrated in Figure 7.2-1. The directives can be recognized as strings of the form `...xxxxxxx` beginning in column one and followed by zero or more parameters. Any line of text which is not a valid directive is treated as a comment and is ignored.

```
; KwikNet Virtual File System created as follows:
;
; Source file                               VFS file name
; -----
; D:\MY_VFS\INDEX.HTM                      \INDEX.HTM
; D:\MY_VFS\HTML\MAIN.HTM                  \HTML\MAIN.HTM
; D:\MY_VFS\HTML\PAGE2.HTM                 \HTML\PAGE2.HTM
; D:\MY_VFS\HTML\PAGE3.HTM                 \HTML\PAGE3.HTM
; D:\MY_VFS\IMAGE\MAIN.GIF                 \IMAGE\MAIN.GIF
; D:\MY_VFS\IMAGE\PICTURE.JPG              \IMAGE\PICTURE.JPG

; Volume information
...BASEDIR \

; File information
...SRCPATH D:\MY_VFS
...OUTFILE D:\MY_VFS\APP_VFS.C
...COMPRESS C+
...IN INDEX.HTM
...IN HTML\MAIN.HTM
...IN HTML\PAGE2.HTM
...IN HTML\PAGE3.HTM
...COMPRESS C-
...IN IMAGE\MAIN.GIF
...IN IMAGE\PICTURE.JPG
```

Figure 7.2-1 VFS Definition File Sample 1

The directives are almost self explanatory. Directive `...BASEDIR` defines the VFS volume base as `\`. Hence all VFS file names will begin with `\`. The `...OUTFILE` directive indicates that the Virtual File System is to be generated in C source file `APP_VFS.C` in directory `D:\MY_VFS` on the user's development system.

Directive `...SRCPATH` indicates that the source files from which the Virtual File System is to be constructed all reside in directory `D:\MY_VFS` on the user's development system. The `...COMPRESS` and `...IN` directives indicate that the Virtual File System consists of four compressed text files and two uncompressed binary files.

Note that the hierarchy of files in the Virtual File System base directory `\` exactly matches that of the source files in directory `D:\MY_VFS` on the user's development system.

A variation of the example from Figure 7.2-1 will illustrate how easy it is to reorganize files within a Virtual File System. The VFS Definition File shown in Figure 7.2-2 defines a Virtual File System with all files located in the VFS volume's base directory. Note that the same source files used in Figure 7.2-1 are used again.

This example introduces several new features. It shows that the `...SRCPATH` directive can be reused to change the source directory from which source files are retrieved by the VFS Generator.

Note that the directive `...COMPRESS` is not used to disable compression before the image files are added to the Virtual File System. Instead, the switch `c-` is used to disable compression of binary files `MAIN.GIF` and `PICTURE.JPG`.

This example shows that the `...OUTFILE` directive can be used multiple times to split the resulting Virtual File System source code into more than one C file. In this case, the Virtual File System will span two C source files, `MY_HTML.C` and `MY_IMAGE.C`, both in directory `D:\MY_VFS` on the user's development system. This feature can be very useful if the size of the C file needed to describe your Virtual File System is very large.

```

; KwikNet Virtual File System created as follows:
;
; Source file                      VFS file name
; -----
; D:\MY_VFS\INDEX.HTM              \INDEX.HTM
; D:\MY_VFS\HTML\MAIN.HTM          \MAIN.HTM
; D:\MY_VFS\HTML\PAGE2.HTM         \PAGE2.HTM
; D:\MY_VFS\HTML\PAGE3.HTM         \PAGE3.HTM
; D:\MY_VFS\IMAGE\MAIN.GIF         \MAIN.GIF
; D:\MY_VFS\IMAGE\PICTURE.JPG      \PICTURE.JPG

; Volume information
...BASEDIR      \

; File information
...COMPRESS      c+
...OUTFILE       D:\MY_VFS\MY_HTML.C
...SRCPATH       D:\MY_VFS
...IN            INDEX.HTM
...SRCPATH       D:\MY_VFS\HTML
...IN            MAIN.HTM
...IN            PAGE2.HTM
...IN            PAGE3.HTM

...OUTFILE       D:\MY_VFS\MY_IMAGE.C
...SRCPATH       D:\MY_VFS\IMAGE
...IN            MAIN.GIF,c-
...IN            PICTURE.JPG,c-

```

Figure 7.2-2 VFS Definition File Sample 2

File Compression

The VFS Generator can compress the files which make up your Virtual File System, thereby reducing the memory required for your VFS image. The compression is done using a simple text string replacement algorithm. This technique has been adopted because the run-time file expansion is simple and fast. It is also especially suitable for HTML files which contain frequently repeated HTML tags. Although other formal binary compression algorithms may produce better compression, the VFS memory image savings have to be dramatic just to recover the code and data space required by the file expansion process. Furthermore, other compression utilities are available to compress files (such as GIF images) before they are merged into your Virtual File System.

The VFS Generator can only compress the text strings which it finds in its VFS String List. This list of strings can be defined by you in your VFS Definition File or in a separate VFS String File. The KwikNet Virtual File System includes such a VFS String File (*KN_VFG.UVF*) tailored specifically for HTML file compression.

As the VFS Generator encounters string matches during its creation of a VFS volume, it adds the matched string to that volume's VFS Compression Table. Although there is no limit to the number of compression candidate strings in your VFS String List, the VFS Compression Table is limited to a maximum of 127 strings. Once the table is full, compressions will be restricted to those strings already in the table.

Figure 7.2-3 illustrates the directives which are used to define strings for inclusion in the VFS String List. For other examples, review the KwikNet VFS String File *KN_VFG.UVF*.

The *...STR* directive is used to define case sensitive compression strings. Such strings are inserted into the VFS String List exactly as defined.

The *...TAG* directive is used to define compression strings whose case can be easily adjusted using the *...TAGCASE* directive. If the *...TAGCASE* parameter is *UPPER* or *LOWER*, any string defined using the *...TAG* directive will be converted to upper or lower case respectively prior to insertion in the VFS String List. If the *...TAGCASE* parameter is *NONE*, the *...TAG* directive will be treated just like the *...STR* directive. The tag feature allows strings defined using any case, including mixed case, to be easily converted to upper or lower case without having to edit the strings.

An exact match with strings in the VFS String List is required for a string to be compressed. In this example, if a file contained the string "*KwikNet*", the string would not be compressed. All occurrences of string "<TABLE>" would be compressed but any occurrence of string "<TABLEe>" would not.

```
...STR           "KwikNet"
...STR           "\r\n\r\n"
;
...TAGCASE       UPPER
...TAG           "<table>"
...TAG           "<table "
...TAG           "</table>"
```

Figure 7.2-3 VFS Compression String Definition

VFS Definition File Directives

Each directive in a VFS Definition File consists of a keyword of the form `...XXXXXX` in column one followed by zero or more parameters which you must provide. In the examples which follow, symbolic names are used to represent parameters. When you create a VFS Definition File, you must replace the symbolic name with your particular parameter.

Volume Base

The VFS volume base which forms the initial part of every virtual file name must be defined using keyword `...BASEDIR`.

```
...BASEDIR  VOLBASE
```

Parameter `VOLBASE` is the text string to be used as the volume base. The volume base should adhere to the file path naming rules established by the underlying real file system, if any, which you are using with your application. If no real file system is used, it is recommended that you use UNIX path naming rules.

If the volume base string begins with a drive name (such as `C:`), the drive name characters will be stripped from the volume base string if drive names are not supported by the underlying file system. Hence, if your volume base includes a drive name, the resulting VFS volume can be used with any file system, including one that does not support drive names.

If the volume base string does not end with a forward or backward slash, a VFS separator character will be automatically appended.

If the `...BASEDIR` directive is omitted, the volume base `/` will be used.

Source Path

The source path identifies the location of the files which the VFS Generator will put into your Virtual File System. By default, the VFS Generator assumes that such files will be found in the VFS Generator's working directory. The `...SRCPATH` keyword can be used to direct the VFS Generator to access files from some other directory.

```
...SRCPATH  FILE_LOC
```

Parameter `FILE_LOC` must identify a valid file path on the development system on which the VFS Generator is executing. Usually this will be the path to a directory in which you have collected the files which will form your Virtual File System. Note that parameter `FILE_LOC` only specifies where files reside on your development system. It does not influence the structure of your Virtual File System in any way.

Once the source path has been defined, only files within that directory or its subdirectories can be inserted into your Virtual File System. However, the source path can be redefined using the `...SRCPATH` directive as often as required.

If parameter `FILE_LOC` is omitted, the VFS Generator will revert to using the default source path which was in effect when the VFS Generator was started.

Source Files

The files which are to be inserted into your Virtual File System are specified using keyword `...IN`.

```
...IN FILE,COMPRESS
```

Parameter *FILE* is the name of a file to be included in your Virtual File System. The file must be located in a directory which is accessible from the directory specified by the `...SRCPATH` directive. The file name specified by parameter *FILE* can include a relative path. Absolute paths or root relative paths are not allowed.

Files are inserted into your Virtual File System in the order in which the `...IN` directives are listed in your VFS Definition File.

Parameter *COMPRESS* can be used to override the VFS Generator's current compression mode. The parameter only affects the file specified by *FILE*. If parameter *COMPRESS* is set to *c+*, the file will be compressed as it is inserted into your Virtual File System, even if compression is otherwise disabled. If parameter *COMPRESS* is set to *c-*, the file will not be compressed, even if compression is otherwise enabled. If parameter *COMPRESS* is set to *c*, the file will be compressed according to the compression mode which was in effect when the VFS Generator was started.

Output File(s)

By default, your Virtual File System is generated as a C file whose name is derived from the name of your VFS Definition File. For example, if your VFS Definition File is named *YOURVFS.UVF*, then the resulting C file will be named *YOURVFS.C*. The C file is created in the directory in which your VFS Definition File is located.

You can use the `...OUTFILE` keyword to direct the VFS Generator to use a different file name and directory for your Virtual File System C file.

```
...OUTFILE VFSFILE
```

Parameter *VFSFILE* must provide the path and file name of the file to be created by the VFS Generator. The path must be a valid file path on the development system on which the VFS Generator is executing. If no path is provided, the file will be created in the VFS Generator's working directory.

The `...OUTFILE` directive can be used repeatedly to split the resulting Virtual File System source code into more than one C file. Whenever a `...OUTFILE` directive is encountered, the VFS Generator creates a new C file and inserts subsequent source files specified with `...IN` directives into the new file. This feature can be very useful if the size of the C file needed to describe your entire Virtual File System would otherwise be very large.

Compression Mode

The VFS Generator operates with file compression enabled or disabled. The current compression state is referred to as the compression mode. By default, compression is enabled. All files are compressed as they are inserted into your Virtual File System. This default behavior can be altered using a VFS Generator command line switch as described in Chapter 7.3.

The `...COMPRESS` keyword can be used to change the VFS Generator's compression mode.

```
...COMPRESS CMODE
```

Parameter `CMODE` defines the new compression mode. Set `CMODE` to `c+` to enable compression. Set `CMODE` to `c-` to disable compression. Omit parameter `CMODE` to restore the default compression mode which was in effect when the VFS Generator was started.

Compression Strings

Compression strings are defined using either the `...STR` or `...TAG` keyword. These keywords can be used in your VFS Definition File. They can also be used in a separate VFS String File.

```
...STR      "CSTRING"  
...TAG      "CSTRING"
```

Parameter `CSTRING` is the compression string. The string must begin and end with the double quote character (`"`, `0x22`). The string can include any of the printable ASCII characters (`0x20` to `0x7E`) as well as the tab (`HT`, `0x09`), linefeed (`LF`, `0x0A`) and return (`CR`, `0x0D`) characters. To include the following characters, you must use the specified two character escape sequence.

```
HT (0x09)  \t  
LF (0x0A)  \n  
CR (0x0D)  \r  
" (0x22)   \"  
\ (0x5C)   \\  

```

The `...STR` directive is used to define case sensitive compression strings. Such strings are inserted into the VFS String List exactly as defined.

The `...TAG` directive is used to define compression strings whose case can be easily adjusted using the `...TAGCASE` directive. Any string defined using the `...TAG` directive will be converted to upper or lower case or left unaltered prior to insertion in the VFS String List according to the `...TAGCASE` specification.

Tag String Case Adjustment

The `...TAGCASE` keyword is used to adjust the case of all compression strings defined using the `...TAG` keyword. The `...TAGCASE` feature allows strings defined using any case, including mixed case, to be easily converted to upper or lower case without having to edit the strings.

```
...TAGCASE  USECASE
```

If parameter `USECASE` is the word `UPPER` or `LOWER`, any compression string defined using the `...TAG` directive will be converted to upper or lower case respectively prior to insertion in the VFS String List. If parameter `USECASE` is the word `NONE`, compression strings defined using the `...TAG` directive will be inserted in the VFS String List without alteration.

VFS String File

By default, the VFS Generator will search its own VFS String File, `KN_VFG.UVF`, for compression strings. The `...VFSTRING` keyword can be used to specify an alternate VFS String File.

```
...VFSTRING USERFILE
```

Parameter `USERFILE` must provide the path and file name of the alternate VFS String File to be used by the VFS Generator. The path must be a valid file path on the development system on which the VFS Generator is executing. If no path is provided, the file will be assumed to be in the VFS Generator's working directory.

If parameter `USERFILE` is omitted, the VFS Generator will not search any VFS String File for compression strings. In this case, compression strings, if any, must be defined in your VFS Definition File.

VFS Template File

The VFS Generator must have access to its VFS Template File `KN_VFG.CT` in order to produce the C file representation of your Virtual File System. By default, the template is expected to be in the directory containing the VFS Generator utility program. If, for some reason, you choose to rename the template file or move it to some other directory, you can use the `...TEMPLATE` keyword to provide the alternate file name and/or location.

```
...TEMPLATE CTFILE
```

Parameter `CTFILE` must provide the path and file name of the VFS Template File to be used by the VFS Generator. The path must be a valid file path on the development system on which the VFS Generator is executing. If no path is provided, the file will be assumed to be in the VFS Generator's working directory.

Sector Size

Virtual files are stored in character arrays which, for lack of a better term, are called sector arrays. By default, the maximum size of a sector array is 32,768 bytes. The `...SSIZE` keyword can be used to change the maximum size of these sector arrays.

```
...SSIZE    NBYTES
```

Parameter `NBYTES` defines the maximum number of bytes which the VFS Generator will insert into a sector array. The upper limit for this parameter is 32,768 bytes. Setting this parameter to a small value will increase the memory footprint of your Virtual File System unless most of your virtual files are small.

You should only have to use this directive to overcome array size limitations imposed by some very primitive C compilers. The number of bytes stored in a sector array has no correlation with the sector size on any real disk drive.

The sector size, once adjusted, applies to the virtual files which are subsequently specified using the `...IN` keyword. Although the sector size can be adjusted as often as you wish, there is rarely a need to do so.

Multiple Volumes

In rare cases, you may wish to build an application that uses more than one Virtual File System volume. This process is described in Chapter 7.4.

Each Virtual File System volume is given a volume name. The default volume name used by the VFS Generator is `kvfs_primary`. Each additional volume must be given a unique volume name using the `...VOLUME` keyword.

```
...VOLUME    VOLNAME
```

Parameter `VOLNAME` is the volume name to be given to the Virtual File System created from the VFS Definition File containing the `...VOLUME` directive. All volume names must be unique. Your C compiler must be able to create a public variable using the volume name as the name of the variable.

7.3 Using the VFS Generator

The KwikNet VFS Generator is a utility program which will create a Virtual File System for use with the KwikNet TCP/IP Stack. The VFS Generator is provided ready for use on a PC or compatible running the Microsoft® MS-DOS® or Windows® operating system.

When KwikNet is installed on your hard disk, the VFS Generator utility program and its related files are stored in directory *VFSGEN* in your KwikNet installation directory.

File	Purpose
<i>KN_VFG.EXE</i>	KwikNet VFS Generator (utility program)
<i>KN_VFG.CT</i>	KwikNet VFS Template File
<i>KN_VFG.UVF</i>	KwikNet VFS String File

The Virtual File System generation process is illustrated in the block diagram of Figure 7.3-1.

The VFS Generator reads the definition of your Virtual File System from your VFS Definition File, a text file constructed as described in Chapter 7.2. A VFS String List is constructed from the set of compression strings defined in your VFS Definition File or in your optional VFS String File.

The VFS Generator then creates a set of one or more VFS Data Files using its VFS Template File as a model for each. The data files are text files with names as specified by the *...OUTFILE* directives in your VFS Definition File. The VFS Data Files contain the C language statements which, when compiled, will provide the content of your Virtual File System.

The data for the virtual files in your Virtual File System is derived from the VFS source files specified by the *...IN* directives in your VFS Definition File.

Examples

The Virtual File System is delivered with two sets of HTML files and the VFS Definition File needed to convert each set into a VFS Data File. The files are located in directories *VFSGEN\SAMPLE* and *VFSGEN\VFS_INFO* in your KwikNet installation directory. Both sets of files are for use with the KwikNet WEB Sample Program.

File *VFSGEN\SAMPLE\KNWEBVFS.UVF* is the VFS Definition File used to create the VFS Data File *KNWEBVFS.C* which is compiled and linked with the KwikNet WEB Sample Program.

File *VFSGEN\VFS_INFO\VFS_INFO.UVF* is the VFS Definition File used to create an alternate VFS Data File *VFS_INFO.C* for use with the WEB Sample Program. If you browse the HTML document *VFSGEN\VFS_INFO\VFS_USE.HTM*, you will find a complete description of the construction process to be followed to add this VFS to the KwikNet WEB Sample Program.

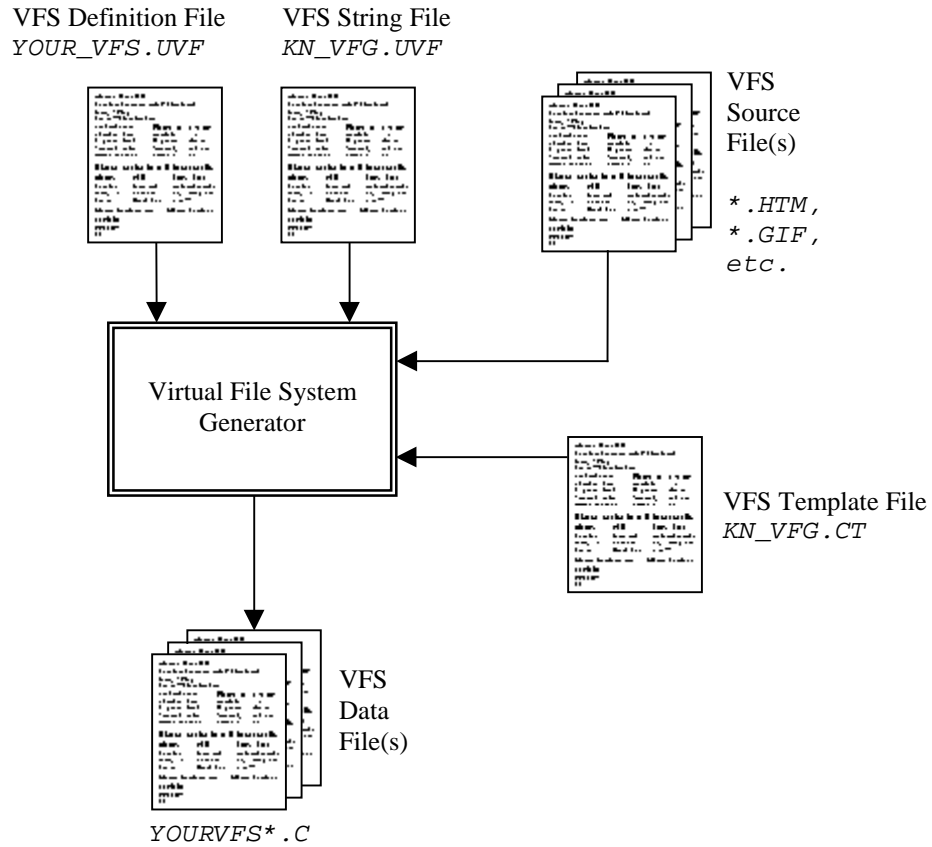


Figure 7.3-1 Using the KwikNet VFS Generator

Running the VFS Generator

The KwikNet VFS Generator is a utility program which is provided ready for use on a PC or compatible running the Microsoft® MS-DOS® or Windows® operating system. The VFS Generator must be started from the MS-DOS command line or from the Windows MS-DOS command prompt. It can also be started from the Windows Run... dialog box. Alternatively, you can create a Windows shortcut to the VFS Generator's filename and then simply double click the shortcut's icon.

The directory which is in effect when the VFS Generator is started is called the **working directory**. Hence, when the VFS Generator is started from an MS-DOS command prompt, the MS-DOS current directory becomes the VFS Generator's working directory. If the VFS Generator is started from a Windows shortcut, the working directory is determined by the start path specified in the shortcut's link file. If the VFS Generator is started from the Windows Run... dialog box, the working directory will be undefined.

The **command line** used to start the VFS Generator is as follows:

```
KN_VFG vsfdefn [-o vfsname] [-p srcpath] [-s vfstring/-s-]  
[-v volname] [-d basedir] [-c+/-c-] [-z ssize] [-t template]  
[-u] [-g/-g+/-g-] [-q/-q0/-q1/-q2/-q3] [-?]
```

The VFS Generator utility program is named *KN_VFG.EXE*.

Only the first parameter, *vsfdefn*, is mandatory. All other parameters are optional as indicated by the enclosing *[]* brackets. Omit the *[* and *]* characters when entering an optional parameter. The symbol */* indicates that only one of the set of optional parameters can be used. Option letters are case sensitive. The order of optional parameters is not important. The default condition for each option is defined in the option descriptions which follow.

If an option is repeated on the command line, an error message may be generated. If no error is observed, then the last (rightmost) use of an option will take precedence.

- | | |
|-------------------|---|
| <i>vsfdefn</i> | Mandatory full path and file name of your VFS Definition File.
A file name with extension <i>UVF</i> , such as <i>MY_VFS.UVF</i> , is recommended.
The working directory is used if no path, or a relative path, is specified. |
| <i>-o vfsname</i> | <i>vfsname</i> is the full path and file name for the generated VFS Data File.
The working directory is used if no path, or a relative path, is specified.
By default, the output VFS Data File will be named to match your definition file <i>vsfdefn</i> but with the file extension changed to <i>.C</i> .
You can use the <i>...OUTFILE</i> directive in your definition file to override either the default or the <i>vfsname</i> output file name. |

- p srcpath* *Srcpath* is the full path defining the location of VFS source files which are to be placed into the Virtual File System being generated.
The working directory is used if a relative path is specified.
The working directory is used if this option is omitted.
You can use the `...SRCPATH` directive in your definition file to override either the default or the *srcpath* path specification.
- s vfstring* *Vfstring* is the full path and file name of your VFS String File.
The working directory is used if no path, or a relative path, is specified.
By default, the VFS String File *KN_VFG.UVF* delivered with KwikNet will be used, provided that it can be found in the directory containing the VFS Generator utility program being executed.
This option, if used, overrides the equivalent parameter, if any, specified in your definition file using the `...VFSTRING` directive.
- s-* The *-s-* option inhibits the use of any VFS String File.
Any `...VFSTRING` directives in your definition file will be ignored.
- v volname* *Volname* is the volume name to be assigned to your Virtual File System.
By default, the volume will be given the name *kvfs_primary*.
All volume names must be unique. Your C compiler must be able to create a public variable with the name *volname*.
This option, if used, overrides the equivalent parameter, if any, specified in your definition file using the `...VOLUME` directive.
- d basedir* *Basedir* defines the volume base for each virtual file in the Virtual File System being generated.
The default volume base is `/`.
This option, if used, overrides the equivalent parameter, if any, specified in your definition file using the `...BASEDIR` directive.
- c+* By default, the VFS Generator operates with file compression enabled.
All files which are placed into the Virtual File System are compressed using a text compression algorithm.
This default mode of operation is equivalent to using the *-c+* option.
- c-* The *-c-* option disables file compression.
You can use the `...COMPRESS` directive in your definition file to override the compression mode established via the command line.
You can also use the *c*, *c+* or *c-* parameter with the `...IN` directive in your definition file to adjust the compression mode used for a specific VFS source file.
- z ssize* Set the VFS virtual file sector size to *ssize* bytes.
The default sector size is 32,768 bytes.
You can use the `...SSIZE` directive in your definition file to override either the default or the *ssize* sector size.

- t template* *Template* is the full path and file name of the KwikNet VFS Template File. The working directory is used if no path, or a relative path, is specified. By default, the VFS Template File *KN_VFG.CT* delivered with KwikNet will be used, provided that it can be found in the directory containing the VFS Generator utility program being executed. This option, if used, overrides the equivalent parameter, if any, specified in your definition file using the *...TEMPLATE* directive.
- u* If files are compressed as they are placed into the Virtual File System, the VFS Generator can produce a summary of the number of times each string in the VFS Compression Table has been used. By default, no summary is generated. The *-u* option forces the VFS Generator to add this summary within a C comment block in the VFS Data File which contains the VFS Compression Table. The summary will therefore be found in the last VFS Data File produced by the VFS Generator.
- g* The *-g* option controls the case conversion to be used for all compression strings defined in your VFS Definition File or in your VFS String File using the *...TAG* directive. Such strings are called tag strings. The *-g* option specifies that tag strings are not to be converted. The *-g+* option specifies that tag strings are to be converted to upper case. The *-g-* option specifies that tag strings are to be converted to lower case. By default, the *-g* option is used. This option, if used, overrides the equivalent parameter, if any, specified in your definition file using the *...TAGCASE* directive.
- qN* The *-qN* option controls the display of messages by the VFS Generator. The *-q* or *-q0* option inhibits the display of any messages. The *-q1* option enables the display of error messages only. The *-q2* option enables the display of error messages and simple progress messages. The *-q3* option enables the display of error messages and more detailed progress messages. By default, the *-q3* option is used.
- ?* Use the *-?* option to display a helpful summary of the command line syntax and available options.

Compiling the VFS Data Files

The VFS Generator produces a set of one or more VFS Data Files which, collectively, form your Virtual File System. These data files are C source files which must be compiled to create your VFS object modules. If you wish, you can combine these VFS object modules into a VFS library module using your object module librarian. Either the VFS object modules or your VFS library module must be linked with your KwikNet application.

Each VFS Data File is compiled in exactly the same manner as your own application modules which use KwikNet. The process is described in Chapter 3.4.

Note that the compilation depends on KwikNet header file *KN_VFS.H* from subdirectory *TCPIP* in the KwikNet installation directory. This header file will be copied to the Treck installation directory *TRECK\INCLUDE* when your KwikNet Library is created. Hence it is always accessible for compilation along with all other KwikNet header files.

Linking with the Virtual File System

Be sure to configure your KwikNet Library to include support for the Virtual File System. To do so, check the Use VFS option on the File System property page as described in Appendix C.2. If any of the files in your Virtual File System have been compressed, be sure to check the option labeled Enable VFS Compression to ensure that these files will be decompressed when accessed by your application.

If you are linking your Virtual File System as a set of one or more VFS object modules, insert them prior to all libraries.

If you are linking your Virtual File System as one or more VFS library modules, insert them following the KwikNet Library *KNnnnIP.LIB*. Note that library files may have the extension *.A* or some other extension as dictated by the toolset which you are using.

Note

If symbol *kvfs_primary* is undefined, you have forgotten to link the VFS data files which make up your Virtual File System. If you have more than one VFS volume, you may have linked the secondary volumes but the primary volume has not been linked.

7.4 Multiple VFS Volumes

The KwikNet Virtual File System usually consists of only one volume. However, the Virtual File System can be extended to include additional volumes. In fact, since virtual files cannot be dynamically created, adding a volume is the only way that a virtual file can be added to the file system at run time.

Do not be fooled by the fact that one volume can be represented by several VFS Data Files. The fact that the virtual files reside in more than one VFS Data File does not alter the fact that, collectively, they make up a single VFS volume.

A volume is a Virtual File System constructed using the VFS Generator. By default, the VFS Generator assigns the volume name *kvfs_primary* to each volume which it creates, making that volume a primary VFS volume. You must have one, and only one, primary VFS volume. Any additional VFS volumes are called secondary volumes.

When using the VFS Generator to create a secondary volume, you must use the *-v* command line option or the *...VOLUME* directive in your VFS Definition File to give your secondary volume a unique name. You can create as many secondary volumes as you require as long as the volume name of each volume is unique.

When KwikNet is started, it initializes the Virtual File System if it is included in your KwikNet Library. Only the primary VFS volume is made ready for use. It is up to your application to add your secondary VFS volumes after KwikNet is operational.

Your application can add a secondary volume to the Virtual File System at run time using VFS service procedure *kvf_voladd()*. Your secondary volume is added at the end of all other volumes present in the Virtual File System at the time of addition. Note that virtual files are accessed according to the order of the volumes which contain them. Hence, a virtual file in the primary volume will take precedence over another virtual file with the same name appearing in a secondary volume.

Your application can also delete a secondary volume from the Virtual File System at run time using VFS service procedure *kvf_voldel()*.

Warning

KwikNet and your application tasks must NOT be using the Virtual File System when your application attempts to add or delete a secondary volume.

7.5 VFS Service Procedures

The following list summarizes KwikNet VFS service procedures which are accessible to your application. These procedures are all present in the KwikNet Library. They are grouped functionally for easy reference.

<i>kvf_close</i>	Close a virtual file
<i>kvf_fsize</i>	Fetch the size of a virtual file
<i>kvf_isvfile</i>	Determine if a file is a virtual file
<i>kvf_open</i>	Open a virtual file
<i>kvf_read</i>	Read from a virtual file
<i>kvf_seek</i>	Seek within a virtual file
<i>kvf_tell</i>	Determine the current position of the file pointer in a virtual file
<i>kvf_voladd</i>	Add a VFS volume
<i>kvf_voldel</i>	Delete a VFS volume

A description of each of these VFS service procedures follows. The descriptions are ordered alphabetically for easy reference and adhere to the style outlined in Chapter 4.6. All of the VFS procedures are described using the C programming language.

kvf_close

kvf_close

Purpose **Close a Virtual File**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_FILES.H*.

```
#include "KN_LIB.H"
#include "KN_FILES.H"
int kvf_close(KN_VFILE *vfd);
```

Description *vfd* is a virtual file descriptor identifying the virtual file to be closed.

Returns If successful, a value of *0* is returned and the file is closed.

 On failure, a value of *-1* is returned.

Note If the parameter *vfd* does not reference a currently open virtual file, the request is passed on via the KwikNet Universal File System Interface to the underlying real file system, if one exists. In such a case, the return value will be dictated by the equivalent *fclose()* function.

See Also *kvf_open()*

Purpose **Fetch the Size of a Virtual File****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_FILES.H*.

```
#include "KN_LIB.H"
#include "KN_FILES.H"
long kvf_fsize(const char *name);
```

Description *Name* is a filepath string providing the full path and file name of the virtual file of interest. A virtual filename must begin with the volume base assigned to a Virtual File System volume. The filepath must not be ambiguous.**Returns** If successful, a value of *n* is returned where *n* is the size of the virtual file measured in bytes. If the virtual file has been compressed, the return value *n* is the size of the expanded file. Note that a virtual file can be of length 0.

If the virtual file specified by parameter *name* does not exist and there is no underlying file system to handle the request, a value of *-1L* is returned.

Note If the parameter *name* does not reference a virtual file, the request is passed on to the KwikNet Universal File System Interface. If an underlying real file system exists, the file size is derived using the equivalent of the *fopen()*, *fseek()* and *ftell()* functions. If, for any reason, the file size cannot be derived, a value of *-1L* is returned.**See Also** *kvf_tell()*

Purpose **Determine if a File is a Virtual File**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_FILES.H*.
 `#include "KN_LIB.H"`
 `#include "KN_FILES.H"`
 `int kvf_isvfile(const char *name);`

Description *Name* is a filepath string providing the full path and file name of the virtual file of interest. A virtual filename must begin with the volume base assigned to a Virtual File System volume. The filepath must not be ambiguous.

Returns A non-zero value is returned if the file specified by *name* is a virtual file.

 Otherwise, a value of 0 is returned.

Purpose **Open a Virtual File****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_FILES.H*.

```
#include "KN_LIB.H"
#include "KN_FILES.H"
KN_VFILE *kvf_open(const char *name, const char *mode);
```

Description *Name* is a filepath string providing the full path and file name of the virtual file to be opened. A virtual filename must begin with the volume base assigned to a Virtual File System volume. The filepath must not be ambiguous.*Mode* is a string defining the type of file access which is required. Only the following file access mode strings are allowed. Note that file writes and text mode translations are not supported.

"r"	Open for read (binary mode used)
"rb"	Open for read using binary mode
"rt"	Open for read using text mode (binary mode is used)

Returns If successful, a virtual file descriptor is returned and the file is opened.On failure, a value of *NULL* is returned.**Note** If the parameter *name* does not reference a virtual file, the request is passed on via the KwikNet Universal File System Interface to the underlying real file system, if one exists. In such a case, the return value will be dictated by the equivalent *fopen()* function.**See Also** *kvf_close()*, *kvf_read()*

Purpose **Read from a Virtual File****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_FILES.H*.
 #include "KN_LIB.H"
 #include "KN_FILES.H"
 *size_t kvf_read(void *buf, size_t size, size_t cnt,*
 *KN_VFILE *vfd);*

Description *Buf* is a pointer to storage for the data items to be read from the file.

Size is the size in bytes of each item which is to be read from the file.

Cnt is the number of items to be read from the file.

Vfd is a virtual file descriptor identifying an open virtual file to be read.

Note For best efficiency when reading bytes from a file, set *size* to 1 and *cnt* to the number of bytes to be read.

Returns If successful, a value of *n* is returned where *n* is the number of full size data items read from the file. The value *n* will be less than *cnt* if the end of file is reached before *cnt* items of *size* bytes each can be read. The virtual file pointer is incremented by the number of bytes actually read.

If *size* is 0 or *cnt* is 0, the value 0 is returned and the storage at **buf* is unaltered.

If parameter *vfd* does not reference an open virtual file and there is no underlying file system to handle the request, a value of -1 is returned.

Note If the parameter *vfd* does not reference a currently open virtual file, the request is passed on via the KwikNet Universal File System Interface to the underlying real file system, if one exists. In such a case, the return value will be dictated by the equivalent *fread()* function.

See Also *kvf_close()*, *kvf_open()*

Purpose **Seek Within a Virtual File****Used by** ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure**Setup** Prototype is in file *KN_FILES.H*.

```
#include "KN_LIB.H"
#include "KN_FILES.H"
int kvf_seek(KN_VFILE *vfd, long offset, int mode);
```

Description *vfd* is a virtual file descriptor identifying a currently open virtual file.

Offset is the displacement from the origin specified by parameter *mode* to which the virtual file's data pointer is to be set.

Mode defines the origin from which the seek is to be measured. Parameter *mode* must be one of the following constants. These constants may be defined in the standard C header file *STDIO.H*. If these constants are not already defined, they are defined in header file *KN_FILES.H*.

SEEK_CUR Seek from the current file pointer position.

SEEK_END Seek from the end of the file.

SEEK_SET Seek from the beginning of the file.

Returns If successful, a value of 0 is returned and the virtual file pointer is positioned at *offset* bytes from the origin specified by *mode*. If you try to seek to a position prior to the beginning of the file, the file pointer is set to the beginning of the file. If you try to seek beyond the end of the file, the file pointer is set to the end of the file.

On failure, a value of -1 is returned. Possible reasons for the error are as follows:

Parameter *vfd* does not reference an open virtual file and there is no underlying file system to handle the request.

Parameter *mode* does not specify one of the allowable constants.

You cannot seek at an *offset* other than 0 in a compressed virtual file.

Note If the parameter *vfd* does not reference a currently open virtual file, the request is passed on via the KwikNet Universal File System Interface to the underlying real file system, if one exists. In such a case, the return value will be dictated by the equivalent *fseek()* function.**See Also** *kvf_fsize()*, *kvf_tell()*

Purpose **Determine the Current Position of the File Pointer in a Virtual File**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_FILES.H*.

```
#include "KN_LIB.H"
#include "KN_FILES.H"
long kvf_tell(KN_VFILE *vfd;
```

Description *vfd* is a virtual file descriptor identifying a currently open virtual file.

Returns If successful, a value of *n* is returned where *n* is the position of the file pointer within the virtual file. If the file pointer is at the end of the file, the value *n* will be the number of bytes in the file.

If parameter *vfd* does not reference an open virtual file and there is no underlying file system to handle the request, a value of *-1* is returned.

Note If the parameter *vfd* does not reference a currently open virtual file, the request is passed on via the KwikNet Universal File System Interface to the underlying real file system, if one exists. In such a case, the return value will be dictated by the equivalent *tell()* function.

See Also *kvf_fsize()*, *kvf_seek()*

kvf_voladd **kvf_voldel**

kvf_voladd **kvf_voldel**

Purpose **Add/Delete a Virtual File System Volume**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *KN_FILES.H*.

```
#include "KN_LIB.H"
#include "KN_FILES.H"
int kvf_voladd(const struct knx_vfsvol *volume);
int kvf_voldel(const struct knx_vfsvol *volume);
```

Description *Volume* is pointer to a VFS volume description created by the VFS Generator. The volume description is a public variable whose name is given by the volume name assigned to the VFS volume by the VFS Generator. The variable resides in one of your VFS Data Files.

Parameter *volume* identifies the VFS volume which you wish to add to or delete from your Virtual File System.

Returns If successful, a value of 0 is returned.

On failure, a value of -1 is returned. Possible reasons for the error are as follows:

A volume which has no files cannot be added to the VFS.
A volume which is not currently in the VFS cannot be deleted.

Example

```
/* Add or remove a secondary VFS volume.                */
/* The volume was created by the VFS Generator and was    */
/* given the volume name my_vfs.                          */
/*                                                         */

#include "kn_lib.h"
#include "kn_files.h"

extern const struct knx_vfsvol my_vfs;

int my_volume(int operation)
{
    if (operation)
        return (kvf_voladd(&my_vfs));

    else    return (kvf_voldel(&my_vfs));
}
```


A. Reference Materials and Glossary

A.1 Reference Materials

The following reference books and documents are recommended by KADAK's technical staff as good sources of information about TCP/IP and related protocols. Comer's text provides interesting historical background and a good general introduction to the topics of interest. Siyan's massive document is an excellent TCP/IP handbook.

Books

CARLSON, James [1998], *PPP Design and Debugging*, Addison Wesley Longman, Inc., Reading, Massachusetts.

COMER, Douglas E. [1991], *Internetworking with TCP/IP Volume I, Principles, Protocols and Architectures*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

MUSCIANO, Chuck and KENNEDY, Bill [1997], *HTML: The Definitive Guide, Second Edition*, O'Reilly & Associates, Inc., Sebastopol, California.

PERKINS, David and McGINNIS, Evan [1996], *Understanding SNMP MIBS*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

SIYAN, Karanjit S. [1997], *Inside TCP/IP, Third Edition*, New Riders Publishing, Indianapolis, Indiana.

STALLINGS, William [1999], *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*, Addison Wesley Longman, Inc., Reading, Massachusetts.

Internet Sources

<code>www.rfc-editor.org</code>	RFCs and related documents via web server
<code>comp.protocols.tcp-ip</code>	News group for ongoing TCP/IP discussion
<code>comp.protocols.snmp</code>	News group for ongoing SNMP discussion

This page left blank intentionally.

A.2 KwikNet Glossary

API	An application programming interface defines the method by which a software program can access software components such as procedures in the KwikNet Library.
App-Task	The name given to the body of application code which uses KwikNet services in a single threaded system. The App-Task is any application function (except those executed by an interrupt service routine) which calls KwikNet to perform some operation.
ARP	Address Resolution Protocol: the TCP/IP protocol used to resolve the correlation between an IP address and a physical hardware address such as an Ethernet address.
BOOTP	Boot Protocol: an older protocol used to derive a network IP address during the startup (boot) initialization of an IP stack.
BSD	The University of California, Berkeley refers to its TCP/IP software release as the Berkeley Software Distribution.
Clock Handler	The name given to the procedure which is called by the ISR or ISP root which services the hardware clock interrupt.
Clock Tick	The interrupt generated by a hardware clock.
Conforming ISP	An AMX Interrupt Service Procedure consisting of an ISP root which calls an Interrupt Handler which has the right to make calls to a subset of the KwikNet service procedures.
DHCP	Dynamic Host Configuration Protocol: a protocol used by a DHCP client to derive a network IP address during the startup (boot) initialization of an IP stack. The client uses services provided by a DHCP server elsewhere on the network.
DNS	Domain Name System: the database distributed across all interconnected networks used to map each text-like machine name to its equivalent IP address.
Error Code	A series of signed integers used by KwikNet to indicate error or warning conditions detected by KwikNet service procedures.
Exit Procedure	An AMX or application procedure executed by AMX during the exit phase when an AMX system is shut down.
Fatal Error	A condition detected by KwikNet which is considered so abnormal that to proceed might risk catastrophic consequences.
FIFO	First in, first out. Usually used to refer to the ordering of elements in a queue or linked list.
FTP	File Transfer Protocol: a TCP/IP protocol used for reliably transferring files between two points on a network.

Handle	An identifier assigned by AMX or KwikNet for use by your application to reference a private AMX or KwikNet data item.
ICMP	Internet Control Message Protocol: a component of the IP protocol which handles error and control messages.
Interrupt Handler	An application procedure called from an ISR or ISP root to service an interrupting device.
Interrupt Service Procedure (ISP)	A procedure (in an AMX application) which is executed in response to an external device interrupt request.
Interrupt Service Routine (ISR)	The procedure in any application which is executed in response to an external device interrupt request. In an AMX application, such a procedure is called an ISP.
IP	Internet Protocol: the protocol that defines the delivery of IP datagrams across an internet in a connectionless, best effort fashion.
IP address	A 32-bit number (address) used to identify the interconnection of a host computer to a physical network. IP addresses are easily recognized when expressed using dotted decimal notation in which IP address <code>0x7F000005</code> is written as "127.0.0.5".
ISP	See Interrupt Service Procedure
ISP root	The ISP code fragment (produced by the AMX Configuration Generator) which informs AMX that an interrupt has occurred and calls an application Interrupt Handler.
ISR	See Interrupt Service Routine
KwikNet Task	The private KwikNet procedure which is responsible for all timing control and event sequencing in a KwikNet application.
Library Header File	A C header file produced during the KwikNet Library construction process and used to compile all KwikNet source modules.

MAC	Media Access Control: a general term used to define the method by which access to a physical network is controlled. This term is sometimes used when referencing Ethernet cards since Ethernet is a very common MAC protocol.
Memory Block	A portion of a memory pool that has been allocated for use by one or more tasks.
Memory Pool	A collection of memory sections whose use is controlled by the AMX Memory Manager.
Memory Pool Id	The handle assigned to a memory pool by AMX for use as a unique memory pool identifier.
Memory Section	A contiguous region of memory assigned to the AMX Memory Manager for allocation to application tasks.
MIB	Management Information Base: the set of variables which constitute a database maintained by a network host which supports SNMP.
Multi-homed	A host computer with interconnections to multiple physical networks is said to be multi-homed.
Multitasking	A method of program execution in which an operating system makes it appear as though several procedures (called tasks) are running concurrently.
Network Parameter File	A text file which can be edited by the KwikNet Configuration Builder to describe a particular KwikNet Library configuration and the networks and device drivers required for a particular KwikNet application.
Network Tick	A multiple of the system tick from which the fundamental KwikNet unit of time is derived. All KwikNet time intervals in the system are measured in multiples of the network tick.
OS	A short form for the two words operating system. When the term OS is used alone, no assumption about the operational characteristics of the OS can be made.
PING	Packet InterNet Groper: the name given to the process of sending an ICMP echo request in order to learn if a destination is reachable.
PPP	Point to Point Protocol: a network protocol used to control the delivery of IP datagrams between two hosts interconnected by a serial link.

RAM	Alterable memory used for data storage and stacks.
Restart Procedure	An AMX or application procedure executed by AMX during the initialization phase when an AMX system is started.
ROM	Read only memory of all types including PROMs, EPROMs and EAROMs.
RTOS	A short form for referencing a real-time operating system, usually one with multitasking capability and reasonably good response to rapidly occurring external events.
RT/OS	The KwikNet syntax used to specify a general purpose operating system. The term RT/OS is used to encompass both multitasking and single threaded operating systems when the distinction is irrelevant.
Semaphore	A data structure which can be used by an RTOS to provide an event signaling mechanism or mutually exclusive access by tasks to specific resources.
Single threaded	A method of program execution in which all procedures execute sequentially. This method of operation is sometimes called single tasking. Although interrupts can cause a brief digression in the program's sequential operation, execution after an interrupt has been serviced always resumes in the procedure which was preempted by the interrupt.
SLIP	Serial Line Internet Protocol: a simple network protocol used to control the delivery of IP datagrams between two hosts interconnected by a serial link.
SMTP	Simple Mail Transfer Protocol: a TCP/IP protocol used to transfer mail messages from one machine to another.
SNMP	Simple Network Management Protocol: a protocol used to monitor and manage the operation of a host computer and the networks to which it is attached. Operations depend on the ability to access and modify variables in the host's Management Information Base (MIB).

System Configuration Module

A software module, produced by the AMX Configuration Builder, which defines the characteristics of a particular AMX application.

System Tick

A multiple of the hardware clock tick from which the fundamental unit of time for an RT/OS is derived. All time intervals in the system are measured in multiples of the system tick.

Tag

A 4-character name that can be assigned to any AMX system data structure when it is created. A tag can be used to find the identifier of a task, timer, semaphore, event group, mailbox, message exchange, memory pool or buffer pool with a particular name.

Tags are also used to identify KwikNet network interfaces and their associated device drivers. KwikNet tags are 1 to 7 character names.

Tailoring file

A special make file included by the make file which is used to build a library or application. The tailoring file provides macro definitions and implicit rules which specify how the make utility can use a specific set of software development tools (compiler, assembler, librarian, linker/locator).

Target Configuration Module

A software module, produced by the AMX Configuration Builder, which defines the characteristics of your target hardware as used in a particular AMX application.

Task

An application procedure which is executed by an RTOS in a way which makes it look as though all such procedures are executing at once.

Task Id

The handle assigned to a task by KwikNet for use as a unique task identifier.

Task Priority

The priority at which a task executes.

TCP

Transport Control Protocol: the protocol used to provide reliable, full-duplex delivery of data streams across a logical connection established between two end points.

Timer

A facility provided by AMX to permit precise interval measurement in AMX applications.

Timer Id

The handle assigned to a timer by AMX for use as a unique timer identifier.

Timer Procedure

An application procedure which is executed by AMX whenever the corresponding timer interval expires.

UDP

User Datagram Protocol: a protocol which permits applications to send and receive datagrams using only the underlying IP network services. UDP datagrams use a port number, in addition to the IP address, to identify the source and destination of each datagram.

This page left blank intentionally.

B. KwikNet Error Codes

TCP/IP Socket Error Codes

TCP/IP socket error codes are signed integers. An error code of *-1* indicates that a socket error has occurred. Codes greater than zero describe the reason for the error. To assist you during testing, the hexadecimal value of the least significant 16-bits of the error code is listed as it might appear in a register or memory dump.

Mnemonic	Value (dec)	Value (hex)	Meaning
	0	0x0000	Socket call successful
<i>KN_SOCKERR</i>	-1	0xFFFF	Socket call failed (Use <i>kn_errno()</i> to fetch reason)
<i>EPERM</i>	201	0x00C9	Permanent error
<i>EBADF</i>	209	0x00D1	The socket descriptor <i>s</i> is invalid (Use <i>KN_EBADF</i> if <i>EBADF</i> conflicts with C)
<i>ENOMEM</i>	212	0x00D4	No memory available
<i>EINVAL</i>	222	0x00DE	Invalid parameter
<i>EMFILE</i>	224	0x00E0	No sockets available
<i>EWOULDBLOCK</i>	235	0x00EB	Caller would block
<i>EINPROGRESS</i>	236	0x00EC	Operation is in progress
<i>EALREADY</i>	237	0x00ED	Operation is already in progress
<i>EDESTADDRREQ</i>	239	0x00EF	Destination address is missing
<i>EMSGSIZE</i>	240	0x00F0	Invalid message size
<i>EPROTOTYPE</i>	241	0x00F1	Protocol type is not supported
<i>ENOPROTOOPT</i>	242	0x00F2	The option is unknown for this protocol
<i>EPROTONOSUPPORT</i>	243	0x00F3	Socket type and/or protocol not supported
<i>EOPNOTSUPP</i>	245	0x00F5	Operation not supported
<i>EAFNOSUPPORT</i>	247	0x00F7	Address family is not supported
<i>EADDRINUSE</i>	248	0x00F8	The specified address is already in use
<i>EADDRNOTAVAIL</i>	249	0x00F9	The specified address is not available
<i>ECONNABORTED</i>	253	0x00FD	The connection was aborted
<i>ECONNRESET</i>	254	0x00FE	The connection has been reset
<i>ENOBUFS</i>	255	0x00FF	No memory buffers are available
<i>EISCONN</i>	256	0x0100	The socket is already connected
<i>ENOTCONN</i>	257	0x0101	The socket is not connected
<i>ESHUTDOWN</i>	258	0x0102	Connection has been shut down
<i>ETIMEDOUT</i>	260	0x0104	Operation timed out
<i>ECONNREFUSED</i>	261	0x0105	The connection was refused
<i>EHOSTUNREACH</i>	265	0x0109	Host destination address is not reachable

KwikNet Error Codes

KwikNet error codes are signed integers. Codes less than zero are error codes. Codes greater than zero are warning codes. To assist you during testing, the hexadecimal value of the least significant 16-bits of the error code is listed as it might appear in a register or memory dump.

Mnemonic	Value (dec)	Value (hex)	Meaning
<i>KN_EROK</i>	0	0	Call successful
Warnings			
<i>KN_WRPROGRESS</i>	1	0x0001	Operation is in progress (not complete)
Device Driver errors			
<i>KN_DER_BADID</i>	-1	0xFFFF	Invalid device id
<i>KN_DER_BADPARAM</i>	-2	0xFFFE	Invalid parameter
<i>KN_DER_BADMODE</i>	-3	0xFFFD	Invalid in current operating mode
<i>KN_DER_NOSUPPORT</i>	-4	0xFFFC	Ioctl command is not supported
<i>KN_DER_DEVICE</i>	-5	0xFFFB	Device specific error occurred
<i>KN_DER_NOTAVAIL</i>	-6	0xFFFA	Requested data not yet available
<i>KN_DER_OVERRUN</i>	-7	0xFFFF9	Receive buffer overflow detected
Programming errors			
<i>KN_ERPARAM</i>	-10	0xFFFF6	Invalid parameter
<i>KN_ERLOGIC</i>	-11	0xFFFF5	Unexpected sequence of events
<i>KN_ERREJECT</i>	-12	0xFFFF4	Requested operation rejected
System errors			
<i>KN_ERNOMEM</i>	-20	0xFFEC	Memory not available for allocation
<i>KN_ERNOBUFFER</i>	-21	0xFFEB	Packet buffer not available
<i>KN_ERQUEUE</i>	-22	0xFFEA	Queuing resource not available
<i>KN_ERTIMEOUT</i>	-23	0xFFE9	Operation timeout
<i>KN_ERNETWORK</i>	-24	0xFFE8	Error generated by network stack
<i>KN_ERIFACE</i>	-25	0xFFE7	Cannot find a device interface

KwikNet Error Codes (continued)

Mnemonic	Value (dec)	Value (hex)	Meaning
TFTP errors			
<i>KN_ERTFD</i>	-260	0xFEFC	Invalid TFTP descriptor
<i>KN_ERTFNOCONN</i>	-261	0xFEFB	Not connected
<i>KN_ERTFEOF</i>	-262	0xFEFA	Not at end of file
<i>KN_ERTFPROTO</i>	-263	0xFE99	TFTP error packet was sent or received
<i>KN_ERTFUDP</i>	-264	0xFE98	Unknown UDP error
<i>KN_ERTFWOULDBLOCK</i>	-265	0xFE97	Operation would have blocked
<i>KN_ERTFTRUNC</i>	-266	0xFE96	Data truncated to fit buffer size
<i>KN_ERTFDTYPE</i>	-267	0xFE95	Invalid operation descriptor type
<i>KN_ERTFSTATE</i>	-268	0xFE94	Invalid operation for current state
<i>KN_ERTFTIMEOUT</i>	-269	0xFE93	Operation timed out
<i>KN_ERTFNOPORT</i>	-270	0xFE92	No local UDP ports available
Telnet errors			
<i>KN_ERTELND</i>	-240	0xFF10	Invalid Telnet descriptor
<i>KN_ERTELSOCK</i>	-241	0xFF0F	Telnet socket fault
<i>KN_ERTELNOCONN</i>	-242	0xFF0E	No connection with Telnet peer
<i>KN_ERTELNOCBF</i>	-243	0xFF0D	No callback function provided
<i>KN_ERTELREJECT</i>	-244	0xFF0C	Invalid operation for Telnet entity
<i>KN_ERTELINCBF</i>	-245	0xFF0B	Invalid request by callback function
<i>KN_ERTELNOSESS</i>	-246	0xFF0A	Operation requires a Telnet session
<i>KN_ERTELINSESS</i>	-247	0xFF09	Operation invalid in Telnet session
<i>KN_ERTELNOSPC</i>	-248	0xFF08	No buffer space available
<i>KN_ERTELBUSY</i>	-249	0xFF07	Busy: command send in progress
<i>KN_ERTELNOLOG</i>	-250	0xFF06	Logging not enabled
<i>KN_ERTELACTIVE</i>	-251	0xFF05	Server has active sessions
<i>KN_ERTELMAXNC</i>	-252	0xFF04	Server has max number of connections

KwikNet Error Codes (continued)

Mnemonic	Value (dec)	Value (hex)	Meaning
SMTP errors			
<i>KN_ERSMND</i>	-280	0xFEE8	Invalid SMTP descriptor
<i>KN_ERMSOCK</i>	-281	0xFEE7	Cannot allocate TCP socket
<i>KN_ERSMNET</i>	-282	0xFEE6	Network error (socket or TCP)
<i>KN_ERSMNOCONN</i>	-283	0xFEE5	No SMTP connection exists
<i>KN_ERSMNOSESS</i>	-284	0xFEE4	No client session active
<i>KN_ERSMINSESS</i>	-285	0xFEE3	Active session(s) present
<i>KN_ERSMACTIVE</i>	-286	0xFEE2	SMTP server already active
<i>KN_ERSMFAIL</i>	-287	0xFEE1	Server failed to start or stop
<i>KN_ERSMMODE</i>	-288	0xFEE0	String mode is invalid
<i>KN_ERSMNOLOG</i>	-289	0xFEDF	Logging is not enabled
<i>KN_ERSMREFUSE</i>	-290	0xFEDE	Request refused: operation prohibited
<i>KN_ERSMREJECT</i>	-291	0xFEDD	Request rejected by server
<i>KN_ERSMSEQ</i>	-292	0xFEDC	Command out of sequence
<i>KN_ERSMSIZE</i>	-293	0xFEDB	Line too long
<i>KN_ERSMTM_DATA</i>	-294	0xFEDA	Timeout waiting for initial data reply
<i>KN_ERSMTM_SEND</i>	-295	0xFED9	Timeout waiting to send data
<i>KN_ERSMTM_EOM</i>	-296	0xFED8	Timeout waiting for final data reply
<i>KN_ERSMTMCONN</i>	-297	0xFED7	Timeout waiting for connection reply
<i>KN_ERSMTMREPLY</i>	-298	0xFED6	Timeout waiting for a server reply

KwikNet Fatal Error Codes

Mnemonic	Value (dec)	Value (hex)	Meaning
<i>KN_FERNOTASK</i>	100	0x0064	Cannot find KwikNet Task
<i>KN_FERNOTMR</i>	101	0x0065	Cannot create network timer
	102	0x0066	reserved
	103	0x0067	reserved
<i>KN_FERFN</i>	104	0x0068	Cannot send fn msg to KwikNet Task
	105	0x0069	reserved
<i>KN_FEREVSIG</i>	106	0x006A	Cannot send event msg to KwikNet Task
<i>KN_FERSUSPEND</i>	107	0x006B	Cannot suspend a task to wait for event
<i>KN_FERRESUME</i>	108	0x006C	Cannot resume a task after event occurs
	109	0x006D	reserved
<i>KN_FERNOSEM4</i>	110	0x006E	No semaphores available for use
	111	0x006F	reserved
<i>KN_FERLOCK</i>	112	0x0070	Resource lock failed
<i>KN_FERUNLOCK</i>	113	0x0071	Resource unlock failed
<i>KN_FERNOMEM</i>	114	0x0072	No memory for allocation
<i>KN_FERBADMEM</i>	115	0x0073	Free memory that was never allocated
<i>KN_FEREXIT</i>	116	0x0074	Cannot send exit to KwikNet Task
<i>KN_FERSTART</i>	117	0x0075	Cannot start KwikNet Stack operation
	118	0x0076	reserved
<i>KN_FERPORT</i>	119	0x0077	Custom port panic
<i>KN_FERPANIC</i>	120	0x0078	KwikNet TCP/IP Stack panic

This page left blank intentionally.

C. KwikNet File System Interface

C.1 Introduction

The KwikNet TCP/IP Stack does not require a file system for normal use. However, several of the optional KwikNet components, such as the FTP client and server and the HTTP Web Server, do require file services. For these options, KwikNet offers a variety of file system solutions, all of which are accessed through the Treck file system application programming interface (API) which is documented in Chapter 6 of the Treck TCP/IP Stack User Manual.

C.1.1 Treck File Systems

The Treck TCP/IP Stack includes three file system variants. The **Treck RAM File System** offers basic file services using a fixed region of memory for file storage. This simple file system may be adequate for many applications which do not require permanent file storage. It is an excellent starting point for testing your application before a real file system is available. For that reason, all of the KwikNet sample programs which require file services use the Treck RAM File System as an example.

For 16-bit segmented architectures, you must be aware that the entire RAM file storage region must reside in one 64Kb segment. This restriction may preclude your use of the Treck RAM File System for all but the simplest of embedded systems.

The **Treck ROM File System** offers basic file services using a constant, predefined set of files located in a fixed region of memory. The files must be created using the Treck ROM FS Builder, a Windows[®] utility provided with the Treck Web Server. The utility operates as described in the Treck Web Server User Manual.

The **Treck DOS File System** interface provides access to the DOS file services on an MS-DOS[®] or Windows[®] platform. KwikNet does not support this file system variant.

If you wish, you can create your own custom file system interface which uses the Treck File System API to access the services available in your file system. Use the Treck source code for the Treck RAM or ROM File System as an example. You may find it just as convenient to use the KwikNet Universal File System interface described in the next section to adapt your file system API for use with KwikNet.

C.1.2 KwikNet Universal File System

KwikNet provides an alternate interface to the file services provided by the file system operating on the target platform. The KwikNet Universal File System (UFS) interface maps the Treck file system API to equivalent UFS services. The UFS can also be used concurrently with the KwikNet Virtual File System described in Chapter 7.

The Universal File System interface provides access to any of the following file systems:

- AMX/FS File System for use with the AMX Multitasking Kernel
- Standard C using the MS-DOS[®] file system
- Custom user defined file system

The AMX/FS File System can be used with AMX and KwikNet on all of the supported target processors. If you are using AMX 86 on a PC compatible platform, you can directly access the MS-DOS file system. A custom file system can also be adapted for use with AMX and KwikNet.

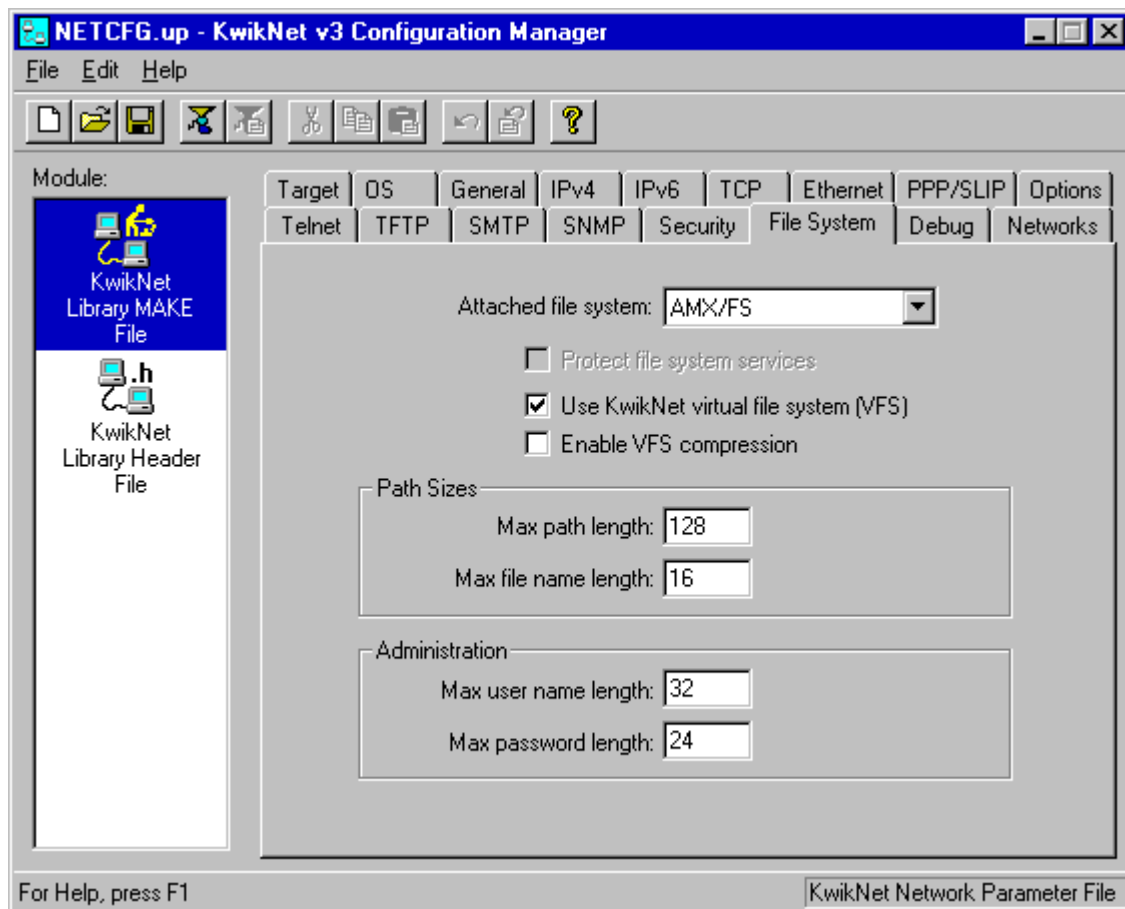
There should be no need to become familiar with the internal operation of the Universal File System unless you must adapt it for use with your own custom file system. This customization procedure is described in Appendix C.5.

Note

You can use a file system accessed via the KwikNet Universal File System (UFS) interface with or without the KwikNet Virtual File System. However, if either is used, you cannot use any of the Treck file systems.

C.2 KwikNet File System Parameters

The KwikNet Universal File System (UFS) interface and the Treck File System variants are included as components in the KwikNet Library. To include the UFS or Treck file system in the library, you must use the KwikNet Configuration Builder to edit your KwikNet Network Parameter File. The file system parameters are edited using the File System property page. The layout of the window is shown below.



File System Parameters (continued)

Attached File System

If you are using a KwikNet option such as FTP or HTTP which requires a file system, choose the file system you will be using with KwikNet.

From the pull down list, select the underlying file system which you intend to use. Choose AMX/FS, "Standard C (MS-DOS)" or "User defined" if you are using a file system which is accessed via the KwikNet Universal File System API. Choose "Treck RAM", "Treck ROM" or "Treck user defined" if you are using one of Treck file systems.

If you are not using any of the optional KwikNet components which require a file system, select None.

Protect File System Services

When operating in a multitasking environment, the file system services must be thread-safe. If the file system you have chosen to use is safe, leave this box unchecked. Otherwise, check this box and KwikNet will use its file system locking mechanism to protect access to the unsafe file system services. Since the AMX/FS and Treck file systems are inherently thread-safe, this option is ignored if any of them is selected.

In a single threaded environment, file system services are inherently thread-safe. Hence, leave this box unchecked.

Use KwikNet Virtual File System

For embedded systems which do not include a file system, KwikNet offers a very simple Virtual File System (VFS) which can provide access to a limited set of read-only files built into the application. The KwikNet Virtual File System is described in Chapter 7.

Check this box if you intend to use the KwikNet Virtual File System. Otherwise, leave this box unchecked.

The Virtual File System can be used without an underlying real file system. The VFS can also be used with any real file system which is accessed via the KwikNet Universal File System (UFS) interface. The VFS will forward file requests which it is not equipped to handle, to the underlying file system, if one exists. The VFS cannot be used with any of the Treck file systems.

Enable VFS Compression

Using the KwikNet VFS Generator, you can create compressed HTML files for use with the KwikNet HTTP Web Server. These files must be decompressed by the KwikNet Virtual File System prior to use by the HTTP Web Server. Check this box if any of the read-only files which you have built into your application have been compressed in this manner. Otherwise, leave this box unchecked.

File System Parameters (continued)

Maximum Path and File Name Lengths

Specify the maximum number of characters which can appear in a path name string used to reference a directory location. The path length must include room for a terminating '\0' character. The path length excludes any file name.

Specify the maximum number of characters which can appear in a file name string used to identify any file. The file name includes the base name and extension(s) and any separating characters. The file name length must include room for a terminating '\0' character. The file name length excludes any path information.

There is no reason to set the path or file name lengths any greater than the maximum allowed by the selected file system. For the KwikNet Virtual File System, a path length of 32 and a file name length of 16 are reasonable. For AMX/FS and MS-DOS file systems, a path length of 128 and a file name length of 16 will be adequate.

To minimize memory waste in embedded applications in which short paths are the norm, the maximum path length can be reduced. However, be aware that KwikNet FTP and HTTP servers may be forced to reject requests for service if the paths and file names which they receive exceed the limits defined by these configuration parameters.

Note

The path and file name lengths are only used by the KwikNet Universal File System (UFS) interface, the KwikNet Virtual File System and the KwikNet administration interface.

These parameters are **not used** by any of the Treck file systems. The Treck RAM File System restricts paths to 260 characters. Treck ROM File System paths are determined by the Treck ROM FS Builder.

Administration Parameters (see Appendix D)

C.3 Using the AMX/FS File System

The KwikNet Universal File System (UFS) interface supports the AMX/FS File System. No customization is required. Any file devices, including RAM drives, hard drives, floppy drives and custom devices, which have been attached to the AMX/FS File System are accessible through the UFS interface.

To use the AMX/FS File System with KwikNet, you must first install AMX and AMX/FS and test them in your intended target environment. The AMX/FS Sample Program offers a good starting point. Once you have it operating with your target hardware, you are ready to merge it with KwikNet.

AMX System Configuration

The first step is to merge the AMX configuration information required for both AMX/FS and KwikNet into a new AMX configuration suitable for use with both products. You may have to increase the maximum number of tasks, timers and semaphores to meet the expanding requirements. Adjust the parameters in your AMX User Parameter File accordingly.

If you are using options such as the KwikNet FTP client or server or the KwikNet HTTP Web Server, be sure to account for the total number of client and server tasks which you intend to employ.

Since each server or client task must have file access, you may have to increase the maximum number of tasks which are permitted to concurrently use AMX/FS. You may also have to increase the maximum number of files which can be concurrently in use. Adjust the parameters in your AMX User Parameter File accordingly.

Your AMX configuration must include the device drivers for both the KwikNet network interfaces and the AMX/FS file devices. You may have to adjust your choices of interrupt assignments to prevent conflicts among these devices.

Once you have an integrated AMX configuration, try using it to confirm that AMX/FS works well in the presence of KwikNet but without KwikNet actually in use. Then use the configuration to test that KwikNet will operate on your network without any file operations. Then you are ready to try KwikNet and AMX/FS together.

AMX System Startup

Special care must be taken when launching an AMX system which includes both KwikNet and the AMX/FS File System. Initialize AMX/FS before starting KwikNet. This implies that execution of AMX/FS Restart Procedure *fj_restart()* must precede the call to KwikNet function *kn_enter()*.

AMX/FS requires that a logical drive be mounted before it can be accessed. This operation is not supported by the Universal File System interface. Hence, before starting any KwikNet clients or servers which require file support, your application must call AMX/FS procedure *fjdrvopen()* to mount each of the logical drives which these KwikNet components are permitted to access.

There is no strict rule governing when logical drives should be mounted. The simplest solution is to have an application task unconditionally mount all available logical drives and then start KwikNet. In this way, all KwikNet components which require file support will have access to all logical drives whenever required.

Alternate solutions may better meet your needs. For example, suppose you intend to have one FTP server task which only requires access to logical drive *D:*. Your FTP server task can mount drive *D:* and then begin operating as an FTP server. Note that other tasks are not precluded from accessing drive *D:*. The FTP server task is also not prevented from accessing another logical drive mounted by some other task.

AMX System Shutdown

When shutting down your AMX application, you must stop KwikNet before terminating AMX/FS. Hence KwikNet function *kn_exit()* must reach completion prior to the execution of AMX/FS Exit Procedure *fj_exit()*.

End of Line Indication

The use of CR (`'\r'`, ASCII `0x0D`), LF (`'\n'`, ASCII `0x0A`) or CRLF (CR followed by LF) as an end of line indicator in text files depends on the interpretation (translation) of strings by file streaming functions such as *fread()* and *fwrite()*.

Although the AMX/FS File System is MS-DOS[®] file format compatible, it does not provide a streaming level API. For example, the description of *fjopen()* states that all files are read and written in binary mode only. Hence AMX/FS does no translation of the data, even if the file is opened in text mode.

The Treck file system API supports both binary and record oriented file access. For text transfers, AMX/FS files are read or written in binary mode through a UFS streaming buffer. The file system's CRLF end of line indicator is stripped or added as the text record is transferred to or from users such as an FTP client, FTP server or Web Server. For binary transfers, AMX/FS files are read or written in binary mode and passed unaltered to or from the users of the file system.

C.4 Using the MS-DOS File System

When used with AMX 86, the KwikNet Universal File System (UFS) interface supports the MS-DOS[®] file system. No customization is required. All file devices, including RAM drives, hard drives and floppy drives, are accessible through the UFS interface.

AMX 86 includes a component called the PC Supervisor which permits AMX 86 to be used with MS-DOS on PC platforms. Special care must be taken when using the PC Supervisor with AMX and KwikNet as described in Chapter 3.7.1.

The PC Supervisor Task must be of lower priority than tasks which use it to access MS-DOS. This requirement leads to the following recommended task priority order when using KwikNet components such as an FTP client or server or the HTTP Web Server.

PC Supervisor Clock Tick Task	Highest priority
PC Supervisor Keyboard Task	
KwikNet Task	
FTP server task	
HTTP Web Server task	
FTP client task	
PC Supervisor Task	Lowest priority

Note that the order of priority of servers and clients may have to be adjusted to reflect the relative importance of each of these services in your application.

C.5 Using a Custom File System

The KwikNet Universal File System (UFS) interface can be adapted to support a custom file system. To do so, you need only edit file *KNFSUSER.H* to meet the requirements of your custom file system. No other customization is required.

Once file *KNFSUSER.H* is ready, simply edit the File System parameters in your KwikNet Network Parameter File to reference your custom file system. Then build your KwikNet Library and link it with your application. The KwikNet Universal File System interface will then use your custom file system for all file operations.

File *KNFSUSER.H* serves two purposes. As its name implies, it is a header file which maps all KwikNet file access functions to those in your file system. However, it is also a code generating module which can provide a custom version of any file access function which is not available in your file system. The code generated by this module will actually reside in KwikNet module *KN_FILES.C* which is located in the KwikNet *TCPIP* installation directory.

The following minimal file system services must be provided by your file system.

- Open a file for read, write or append in text or binary mode
- Close a file
- Write *n* elements of size *m* to a file
- Read *n* elements of size *m* from a file
- Seek within a file
- Tell location of file pointer within a file
- Remove (delete) a file
- Flush data to file
- Fetch size of file
- Rename a file

The following file system services, if provided by your file system, will permit the KwikNet FTP server to offer directory services.

- Make a directory
- Remove (delete) a directory
- Verify that a path name references a directory
- Open a directory for listing files
- Close a directory
- Read next file entry in an open directory

This page left blank intentionally.

D. KwikNet Administration Interface

D.1 Introduction

Many network protocols, such as FTP, were originally developed when large mainframe computers were shared by many users. The computers accommodated multiple users, each with a password and all administered by a higher authority. Network protocols were developed to support user names and passwords.

Unfortunately, user name and password administration services are rarely provided by the operating systems found in desktop computers or embedded devices. This is true even of KADAK's AMX Multitasking Kernel.

The KwikNet TCP/IP Stack does not require a user administration system for normal use. However, to accommodate protocols such as FTP and HTTP, KwikNet provides its own user name and password administrative services.

Note

The KwikNet administration interface is only supported if you are using the KwikNet Virtual File System and/or a file system accessed via the KwikNet Universal File System (UFS) interface.

The KwikNet administration interface is **not supported** by any of the Treck file systems.

User Definitions

All KwikNet users are defined in array *kn_users[]* in module *KN_ADMIN.C* located in the KwikNet *TCPIP* directory. Each user definition is a *knx_userinfo* structure which is defined in header file *KN_ADMIN.H* located in the KwikNet *TCPIP* directory.

```
struct knx_userinfo {                /* User info structure */
    int    xu_access;                /* User access rights */
    int    xu_rsv1;                  /* Fill for alignment */
    void    *xu_app;                 /* Reserved for application */
    char    xu_username[KN_FS_LUSER]; /* User name */
    char    xu_password[KN_FS_LPASS]; /* User password */
    char    xu_basedir[KN_FS_LPATH+4]; /* User base directory */
};
```

To add, modify or delete users, you must edit file *KN_ADMIN.C*. Each definition includes a user name, an unencrypted password, a base directory path and a definition of the user's access rights. User names and passwords are character arrays. The base directory defines the path to a file directory considered to be the user's base (home) directory if a file system is employed. A pointer variable in the definition is reserved for the private use of your application.

User Access Rights

User access rights are formed from the logical OR of the following bit masks which are defined in header file *KN_ADMIN.H*.

<i>KN_ADM_ACC_READ</i>	Allow file read
<i>KN_ADM_ACC_WRITE</i>	Allow file write
<i>KN_ADM_ACC_REMOVE</i>	Allow file remove (delete)
<i>KN_ADM_ACC_DIRSEL</i>	Allow directory traversal (selection)
<i>KN_ADM_ACC_DIRLIST</i>	Allow directory listing
<i>KN_ADM_ACC_DIRMK</i>	Allow directory make
<i>KN_ADM_ACC_DIRRM</i>	Allow directory remove (delete)
<i>KN_ADM_ACC_VISIBLE</i>	Allow file visibility
 <i>KN_ADM_ACC_FULL</i>	 Allow full access (all of the above)

These access rights are used by optional KwikNet components, such as the FTP server, to restrict a user's access to directories and files. Access right *KN_ADM_ACC_DIRSEL* is required to be able to change directories. Most of the other access rights are self explanatory.

The visibility right is special. If a user has access right *KN_ADM_ACC_VISIBLE*, then the user will have full view of all files and directories. Without this access right, the user will not be able to view files or traverse directories which are above the user's base directory.

Customizing Administration Services

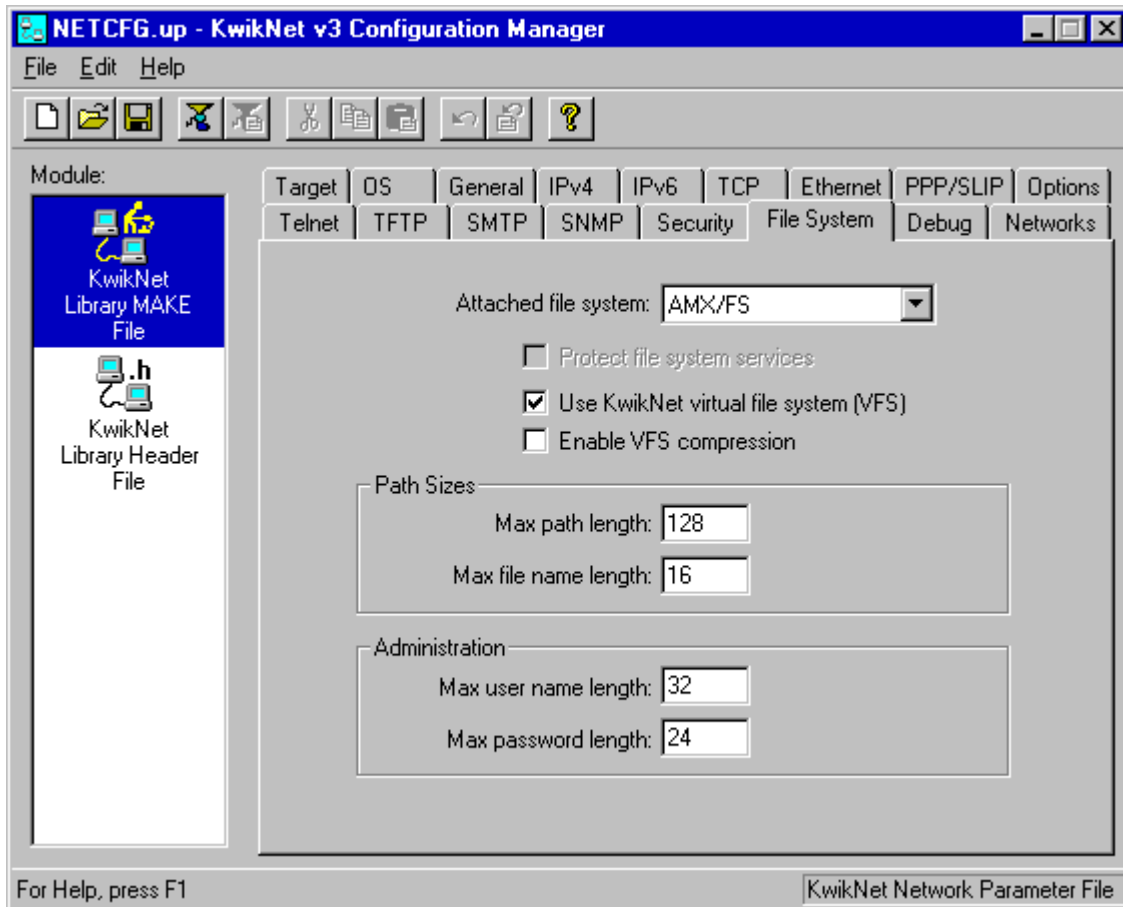
File *KN_ADMIN.C* defines several users. User *anonymous* with no password has read only access to files and directories beginning at a base directory determined by the KwikNet server making use of the user definitions. User *KADAK* with password *KwikNet* has full access to all files from the root directory. You are free to alter these definitions to suit your needs.

File *KN_ADMIN.C* also includes a set of functions which are used by servers, such as the KwikNet FTP server, to validate user names, passwords and access rights. Other functions in the module validate directory path information if a file system is used. Strings of the form ">>>>" identify functions in the file which may require modification. You can alter these validation functions, if necessary, to meet the needs of your application.

After editing the file *KN_ADMIN.C*, you must build your KwikNet Library to incorporate the revised module. Before building the KwikNet Library, be sure to adjust the maximum user name and password lengths (see Appendix D.2) to meet or exceed the lengths of the user names and passwords in your user definitions.

D.2 KwikNet Administration Parameters

The KwikNet administration interface forms part of the KwikNet Library. To adapt the administration interface parameters for your use, you must use the KwikNet Configuration Builder to edit your KwikNet Network Parameter File. The administration parameters are edited using the File System property page. The layout of the window is shown below.



Administration Parameters (continued)

Maximum User Name and Password Lengths

Specify the maximum number of characters which can appear in a user name string. The user name length must include room for a terminating ' \0 ' character.

Specify the maximum number of characters which can appear in a user password string. The password length must include room for a terminating ' \0 ' character.

For most applications, a user name length of 32 and a password length of 24 will be adequate.

To minimize memory waste in embedded applications in which short user names and passwords might be expected, the maximum lengths can be reduced. However, be aware that the KwikNet FTP server may be forced to reject requests for service if the user names and passwords which it receives exceed the limits defined by these configuration parameters.

Note

The KwikNet administration interface is only supported if you are using the KwikNet Virtual File System and/or a file system accessed via the KwikNet Universal File System (UFS) interface.

The user name and password lengths are **not required** if you are using any of the Treck file systems.

File System Parameters (see Appendix C)

E. KwikNet Sample Program Architecture

The manner in which the KwikNet TCP/IP Sample Program starts and operates is completely dependent upon the underlying operating system with which KwikNet is being used. Operation can be either multitasking or single threaded. All sample programs provided with KwikNet and its optional components share the common implementation methodology about to be described.

All KwikNet sample programs are built upon a common framework. The building blocks are a set of files located in toolset directory *TOOLXXX\SAM_COMN* where *XXX* is KADAK's mnemonic for the particular set of software development tools which you are using. The common files and the procedures which they contain are listed in Figure E-1.

A quick review of the common sample program files will indicate that most of the implementation is devoted to the man/machine interface and to the startup process. These two topics always seem to account for the bulk of any networking example, no matter how simple the actual network operations may be.

Console Interface

The KwikNet data logging service, message recording service and console driver have been described in detail in Chapters 1.6 to 1.7 of the KwikNet TCP/IP Stack User's Guide. If you review that material again, you will note that the procedures referenced in the description are all present in the common sample program modules listed in Figure E-1.

The sample programs use KwikNet procedure *kn_dprintf()* to record messages using the KwikNet data logging service. KwikNet passes each such message to the log function *sam_record()* in the Application OS Interface module *KNSAMOS.C*. This common log function is specified on the Debug property page of the Network Parameter File provided with each KwikNet sample program.

The sample programs use console driver procedures *knconin()* and *knconins()* for console input and *knconout()* and *knconouts()* for console output.

The sample programs also use a command line parsing service provided by console driver procedure *kncon_parse()*. This procedure parses a command string into its various tokens according to directions provided by the caller.

The sample programs use a common error message generation service provided by console driver procedure *kncon_error()*. This procedure generates an error message on the console device. The error message is derived from a KwikNet error code using a message list provided by the sample program.

Many of the interactive KwikNet sample programs implement a dump command to display recorded messages. These applications call console driver procedure *kncon_logdump()* to extract all of the message strings from the recording array using recording procedure *kn_loggets()*. The extracted messages are displayed on the console device. Once all recorded strings have been displayed, the recording list is reset with a call to recording procedure *kn_loginit()*.

<i>KNSAMOS.C</i>	Application OS Interface	
	<i>main()</i>	Sample program main entry point
	<i>sam_osshutdown()</i>	OS shutdown on exit
	<i>sam_ostkprep()</i>	Task create/prepare
	<i>sam_ostkstart()</i>	Task start
	<i>sam_record()</i>	Log function used by <i>kn_dprintf()</i>
	<i>print_task()</i>	Print task
	<i>backg_task()</i>	Background task (AMX only)
	<i>rrproc()</i>	Restart Procedure (AMX only)
	<i>exproc()</i>	Exit Procedure (AMX only)
<i>KNRECORD.C</i>	Message recording services	
	<i>kn_loginit()</i>	Initialize data recording services
	<i>kn_logputc()</i>	Record one character
	<i>kn_logmsg()</i>	Record a message
	<i>kn_loggets()</i>	Get one recorded message from the recorded list
<i>KNCONSOL.C</i>	Console driver	
	<i>knconinit()</i>	Initialize console device for use
	<i>knconexit()</i>	Close console device upon exit
	<i>knconin()</i>	Get a character from the console device
	<i>knconins()</i>	Get a string from the console device
	<i>knconout()</i>	Send a character to the console device
	<i>knconouts()</i>	Send a string to the console device
	<i>kncon_logprep()</i>	Prepare to use console for data logging
	<i>kncon_logputc()</i>	Log a character to the console device
	<i>kncon_logdump()</i>	Log all recorded strings on the console device
	<i>kncon_parse()</i>	Parse a console command line
	<i>kncon_error()</i>	Generate an error message on the console device
<i>KN8250S.C</i>	INS8250 (NS16550) UART driver	
	<i>kn_iouart()</i>	All serial I/O device operations

Figure E-1 KwikNet Sample Program Procedures

KwikNet Sample Program Operation with AMX

When KwikNet is used with AMX, the KwikNet sample programs operate as follows. Once your board level initialization is complete and the C startup code has been executed, the sample program begins execution at *main()* in the Application OS Interface module *KNSAMOS.C*.

The **main program** makes a series of calls to initialize the various components which make up the sample program. Your AMX board support function *chbrdinit()* is called to set up the hardware environment for AMX use. KwikNet board driver procedure *kn_brdrreset()* is called to initialize its interrupt support for all KwikNet device drivers.

The KwikNet message recording interface is initialized with a call to *kn_loginit()*. If the console driver has been configured for use as the recording device, procedure *kn_loginit()* calls console driver procedure *kncon_logprep()* to prepare it accordingly.

Next, the *main()* procedure calls *kn_osprep()* in the KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet Library) to initialize the RTOS interface. Since this procedure initializes the KwikNet data logging service, KwikNet procedure *kn_dprintf()* can be used by the sample program even before KwikNet is started.

Finally, the *main()* procedure launches AMX to start the multitasking sample program.

Once AMX is ready, it calls the KwikNet **Restart Procedure** *kn_osready()* in the KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet Library) to initialize the AMX resources required by KwikNet and to prepare the memory allocation subsystem for use by KwikNet and your application.

AMX then calls the application **Restart Procedure** *rrproc()* in the Application OS Interface module *KNSAMOS.C* to start the KwikNet sample program as an AMX application. The AMX clock driver is initialized with a call to procedure *chclockinit()*. Task services in the Application OS Interface module are then used to create and start a low priority background task (procedure *backg_task()*) which provides a simulated software clock in case a real hardware clock is unavailable and an AMX clock driver has not been linked with the sample program.

Next, Restart Procedure *rrproc()* starts the sample program's print task, a task used by some sample programs to log messages on the console device. Finally, procedure *app_prep()* in the sample program module is called to prepare all application level components needed by the sample program.

Every KwikNet sample program provides function *app_prep()* as its advance preparation entry point. This procedure creates and starts one or more application tasks which collectively make up the sample program. One of these tasks, usually called the client task, is the task in charge of the sequence of operations performed by the sample program. For example, the client task often provides a user command line console interface which allows you to interactively control sample program activities.

The **sample program begins** operation at task level once AMX completes its startup processing. The client task executes and calls function *app_enter()*, the entry point to the main body of the sample program.

The client task starts KwikNet with a call to KwikNet procedure *kn_enter()*. KwikNet initializes all of its private resources and starts the KwikNet Task which prepares all network interfaces and their associated device drivers for use.

If the sample program requires AMX/FS file services, procedure *sam_osfsprep()* in the Application OS Interface module *KNSAMOS.C* is called to prepare the AMX/FS File System for use. If the client task provides an interactive user interface, the console driver is initialized with a call to procedure *knconinit()*. Thereafter, the client task orchestrates the sequence of network operations it is designed to illustrate.

The **termination process** is handled by the client task. If the console driver was in use, it is closed with a call to *knconexit()*. KwikNet is stopped with a call to *kn_exit()*. After a brief pause to allow KwikNet to stop, the RTOS shutdown is initiated with a call to *sam_osshutdown()* which simply requests AMX to exit in its usual orderly fashion.

AMX executes the sample program Exit Procedure *exproc()* which calls *chclockexit()* to disable the AMX clock driver. Once all Exit Procedures have been called, AMX ceases operation and returns to the *main()* function from which AMX was launched. One final call to procedure *kn_osfinish()* in the KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet Library) breaks the connection between KwikNet and its RTOS interface.

KwikNet Porting Kit Sample Program - Multitasking Operation

When the KwikNet Porting Kit is used with a multitasking RTOS, the KwikNet sample programs operate as follows. Once your board level initialization is complete and the C startup code has been executed, the sample program begins execution at *main()* in the Application OS Interface module *KNSAMOS.C*.

The **main program** makes a series of calls to initialize the various components which make up the sample program. Your KwikNet board driver procedure *kn_brdreset()* is called to initialize its interrupt support for all KwikNet device drivers.

The KwikNet message recording interface is initialized with a call to *kn_loginit()*. If the console driver has been configured for use as the recording device, procedure *kn_loginit()* calls console driver procedure *kncon_logprep()* to prepare it accordingly.

Next, the *main()* procedure calls *kn_osprep()* in your KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet Library) to initialize your RTOS interface. Since this procedure initializes the KwikNet data logging service, KwikNet procedure *kn_dprintf()* can be used by the sample program even before KwikNet is started. In many cases, procedure *kn_osprep()* will also start your KwikNet clock driver with a call to its initialization procedure *kn_uclockinit()*.

Finally, the *main()* procedure starts your RTOS to run the multitasking sample program. In the example provided with the KwikNet Porting Kit, the RTOS creates a startup task which is executed by the RTOS as it begins operation. The startup task is located at entry point *sam_osmain()* in the Application OS Interface module *KNSAMOS.C*.

Once the RTOS is ready, it executes the startup task procedure *sam_osmain()*. Task services in the Application OS Interface module are used to create and start the sample program's print task, a task used by some sample programs to log messages on the console device. Finally, procedure *app_prep()* in the sample program module is called to prepare all application level components needed by the sample program.

Every KwikNet sample program provides function *app_prep()* as its advance preparation entry point. This procedure creates and starts one or more application tasks which collectively make up the sample program. One of these tasks, usually called the client task, is the task in charge of the sequence of operations performed by the sample program. For example, the client task often provides a user command line console interface which allows you to interactively control sample program activities.

The **sample program begins** operation at task level once the high priority startup task terminates. The client task executes and calls function *app_enter()*, the entry point to the main body of the sample program.

The client task starts KwikNet with a call to KwikNet procedure *kn_enter()*. KwikNet initializes all of its private resources and starts the KwikNet Task which prepares all network interfaces and their associated device drivers for use.

If the client task provides an interactive user interface, the console driver is initialized with a call to procedure *knconinit()*. Thereafter, the client task orchestrates the sequence of network operations it is designed to illustrate.

The **termination process** is handled by the client task. If the console driver was in use, it is closed with a call to *knconexit()*. KwikNet is stopped with a call to *kn_exit()*. After a brief pause to allow KwikNet to stop, the RTOS shutdown is initiated with a call to *sam_osshutdown()* which, if possible, forces the RTOS to terminate execution in an orderly fashion.

If the RTOS ceases operation, it returns to the *main()* function from which it was launched. One final call to procedure *kn_osfinish()* in your KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet Library) breaks the connection between KwikNet and your RTOS. In many cases, procedure *kn_osfinish()* will also stop your KwikNet clock driver with a call to its termination procedure *kn_uclockexit()*.

KwikNet Porting Kit Sample Program - Single Threaded Operation

When the KwikNet Porting Kit is used with a single threaded operating system (OS), the KwikNet sample programs operate as follows. Once your board level initialization is complete and the C startup code has been executed, the sample program begins execution at *main()* in the Application OS Interface module *KNSAMOS.C*.

The **main program** makes a series of calls to initialize the various components which make up the sample program. Your KwikNet board driver procedure *kn_brdreset()* is called to initialize its interrupt support for all KwikNet device drivers.

The KwikNet message recording interface is initialized with a call to *kn_loginit()*. If the console driver has been configured for use as the recording device, procedure *kn_loginit()* calls console driver procedure *kncon_logprep()* to prepare it accordingly.

Next, the *main()* procedure calls *kn_osprep()* in your KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet Library) to initialize your OS interface. Since this procedure initializes the KwikNet data logging service, KwikNet procedure *kn_dprintf()* can be used by the sample program even before KwikNet is started. In many cases, procedure *kn_osprep()* will also start your KwikNet clock driver with a call to its initialization procedure *kn_uclockinit()*.

Finally, the *main()* procedure assumes its **App-Task role** and calls function *app_enter()*, the entry point to the main body of the sample program.

The App-Task starts KwikNet with a call to KwikNet procedure *kn_enter()*. KwikNet initializes all of its private resources and starts the KwikNet Task which prepares all network interfaces and their associated device drivers for use.

If the App-Task provides an interactive user interface, the console driver is initialized with a call to procedure *knconinit()*. Thereafter, the App-Task orchestrates the sequence of network operations it is designed to illustrate.

The **termination process** is handled by the App-Task. If the console driver was in use, it is closed with a call to *knconexit()*. KwikNet is stopped with a call to *kn_exit()*. After a brief pause to allow KwikNet to stop, the OS is forced to shut down with a call to *sam_osshutdown()*. Rarely is any termination processing performed by this function.

The App-Task procedure *app_enter()* returns to the *main()* function from which it was called. One final call to procedure *kn_osfinish()* in your KwikNet OS Interface Module *KN_OSIF.C* (in the KwikNet Library) breaks the connection between KwikNet and your OS. In many cases, procedure *kn_osfinish()* will also stop your KwikNet clock driver with a call to its termination procedure *kn_uclockexit()*.

This page left blank intentionally.